Git Reference

wizardforcel


Published with GitBook

Table of Contents

Git Refernece

From: Git Refernece

Reference

Quick reference guides: GitHub Cheat Sheet (PDF) | Visual Git Cheat Sheet (SVG | PNG)

Setup and Config

- config
- help

Getting and Creating Projects

- init
- clone

Basic Snapshotting

- add
- status
- diff
- commit
- reset
- rm
- mv

Branching and Merging

- branch
- checkout
- merge
- mergetool
- log
- stash
- tag

Sharing and Updating Projects

- fetch
- pull
- push

- remote
- submodule

Inspection and Comparison

- show
- log
- diff
- shortlog
- describe

Patching

- apply
- cherry-pick
- diff
- rebase
- revert

Debugging

- bisect
- blame
- grep

Email

- am
- apply
- format-patch
- send-email
- request-pull

External Systems

- svn
- fast-import

Administration

- clean

- gc
- fsck
- reflog
- filter-branch
- instaweb
- archive
- bundle

Server Admin

- daemon
- update-server-info

Plumbing Commands

- cat-file
- commit-tree
- count-objects
- diff-index
- for-each-ref
- hash-object
- ls-files
- merge-base
- read-tree
- rev-list
- rev-parse
- show-ref
- symbolic-ref
- update-index
- update-ref
- verify-pack
- write-tree

Setup and Config

config

NAME

git-config - Get and set repository or global options

SYNOPSIS

```
git config [<file-option>] [type] [--show-origin] [-z|--null] name [value [value_regex]]
git config [<file-option>] [type] --add name value
git config [<file-option>] [type] --replace-all name value [value_regex]
git config [<file-option>] [type] [--show-origin] [-z|--null] --get name [value_regex]
git config [<file-option>] [type] [--show-origin] [-z|--null] --get-all name [value_regex
git config [<file-option>] [type] [--show-origin] [-z|--null] [--name-only] --get-regexp
git config [<file-option>] [type] [-z|--null] --get-urlmatch name URL
git config [<file-option>] --unset name [value_regex]
git config [<file-option>] --unset-all name [value_regex]
git config [<file-option>] --rename-section old_name new_name
git config [<file-option>] --remove-section name
git config [<file-option>] [--show-origin] [-z|--null] [--name-only] -l | --list
git config [<file-option>] --get-color name [default]
git config [<file-option>] --get-colorbool name [stdout-is-tty]
git config [<file-option>] -e | --edit
```

DESCRIPTION

You can query/set/replace/unset options with this command. The name is actually the section and the key separated by a dot, and the value will be escaped.

Multiple lines can be added to an option by using the *--add* option. If you want to update or unset an option which can occur on multiple lines, a POSIX regexp `value_regex` needs to be given. Only the existing values that match the regexp are updated or unset. If you want to handle the lines that do **not** match the regex, just prepend a single exclamation mark in front (see also EXAMPLES).

The type specifier can be either *--int* or *--bool*, to make *git config* ensure that the variable(s) are of the given type and convert the value to the canonical form (simple decimal number for int, a "true" or "false" string for bool), or *--path*, which does some path expansion (see *--path* below). If no type specifier is passed, no checks or transformations are performed on the value.

When reading, the values are read from the system, global and repository local configuration files by default, and options *--system*, *--global*, *--local* and *--file <filename>* can be used to tell the command to read from only that location (see FILES).

When writing, the new value is written to the repository local configuration file by default, and options *--system*, *--global*, *--file <filename>* can be used to tell the command to write to that location (you can say *--local* but that is the default).

This command will fail with non-zero status upon error. Some exit codes are:

1. The config file is invalid (ret=3),
2. can not write to the config file (ret=4),
3. no section or name was provided (ret=2),
4. the section or key is invalid (ret=1),
5. you try to unset an option which does not exist (ret=5),
6. you try to unset/set an option for which multiple lines match (ret=5), or
7. you try to use an invalid regexp (ret=6).

On success, the command returns the exit code 0.

OPTIONS

--replace-all

Default behavior is to replace at most one line. This replaces all lines matching the key (and optionally the value_regex).

--add

Adds a new line to the option without altering any existing values. This is the same as providing *^$* as the value_regex in `--replace-all`.

--get

Get the value for a given key (optionally filtered by a regex matching the value). Returns error code 1 if the key was not found and the last value if multiple key values were found.

--get-all

Like get, but does not fail if the number of values for the key is not exactly one.

--get-regexp

Like --get-all, but interprets the name as a regular expression and writes out the key names. Regular expression matching is currently case-sensitive and done against a canonicalized version of the key in which section and variable names are lowercased, but subsection names are not.

--get-urlmatch name URL

When given a two-part name section.key, the value for section.<url>.key whose <url> part matches the best to the given URL is returned (if no such key exists, the value for section.key is used as a fallback). When given just the section as name, do so for all the keys in the section and list them.

--global

For writing options: write to global `~/.gitconfig` file rather than the repository `.git/config`, write to `$XDG_CONFIG_HOME/git/config` file if this file exists and the `~/.gitconfig` file doesn't.

For reading options: read only from global `~/.gitconfig` and from `$XDG_CONFIG_HOME/git/config` rather than from all available files.

See also FILES.

--system

For writing options: write to system-wide `$(prefix)/etc/gitconfig` rather than the repository `.git/config`.

For reading options: read only from system-wide `$(prefix)/etc/gitconfig` rather than from all available files.

See also FILES.

--local

For writing options: write to the repository `.git/config` file. This is the default behavior.

For reading options: read only from the repository `.git/config` rather than from all available files.

See also FILES.

-f config-file

--file config-file

Use the given config file instead of the one specified by GIT_CONFIG.

--blob blob

Similar to *--file* but use the given blob instead of a file. E.g. you can use *master:.gitmodules* to read values from the file *.gitmodules* in the master branch. See "SPECIFYING REVISIONS" section in gitrevisions[7] for a more complete list of ways to spell blob names.

--remove-section

Remove the given section from the configuration file.

--rename-section

Rename the given section to a new name.

--unset

Remove the line matching the key from config file.

--unset-all

Remove all lines matching the key from config file.

-l

--list

List all variables set in config file, along with their values.

--bool

git config will ensure that the output is "true" or "false"

--int

git config will ensure that the output is a simple decimal number. An optional value suffix of *k*, *m*, or *g* in the config file will cause the value to be multiplied by 1024, 1048576, or 1073741824 prior to output.

--bool-or-int

git config will ensure that the output matches the format of either --bool or --int, as described above.

--path

git-config will expand leading ~ to the value of *$HOME*, and *~user* to the home directory for the specified user. This option has no effect when setting the value (but you can use *git config bla ~/* from the command line to let your shell do the expansion).

-z

--null

For all options that output values and/or keys, always end values with the null character (instead of a newline). Use newline instead as a delimiter between key and value. This allows for secure parsing of the output without getting confused e.g. by values that contain line breaks.

--name-only

Output only the names of config variables for `--list` or `--get-regexp`.

--show-origin

Augment the output of all queried config options with the origin type (file, standard input, blob, command line) and the actual origin (config file path, ref, or blob id if applicable).

--get-colorbool name [stdout-is-tty]

Find the color setting for `name` (e.g. `color.diff`) and output "true" or "false". `stdout-is-tty` should be either "true" or "false", and is taken into account when configuration says "auto". If `stdout-is-tty` is missing, then checks the standard output of the command itself, and exits with status 0 if color is to be used, or exits with status 1 otherwise. When the color setting for `name` is undefined, the command uses `color.ui` as fallback.

--get-color name [default]

Find the color configured for `name` (e.g. `color.diff.new`) and output it as the ANSI color escape sequence to the standard output. The optional `default` parameter is used instead, if there is no color configured for `name`.

-e

--edit

Opens an editor to modify the specified config file; either --*system*, --*global*, or repository (default).

--[no-]includes

Respect `include.*` directives in config files when looking up values. Defaults to `off` when a specific file is given (e.g., using `--file`, `--global`, etc) and `on` when searching all config files.

FILES

If not set explicitly with --*file*, there are four files where *git config* will search for configuration options:

$(prefix)/etc/gitconfig

System-wide configuration file.

$XDG_CONFIG_HOME/git/config

Second user-specific configuration file. If $XDG_CONFIG_HOME is not set or empty, `$HOME/.config/git/config` will be used. Any single-valued variable set in this file will be overwritten by whatever is in `~/.gitconfig`. It is a good idea not to create this file if you sometimes use older versions of Git, as support for this file was added fairly recently.

~/.gitconfig

User-specific configuration file. Also called "global" configuration file.

$GIT_DIR/config

Repository specific configuration file.

If no further options are given, all reading options will read all of these files that are available. If the global or the system-wide configuration file are not available they will be ignored. If the repository configuration file is not available or readable, *git config* will exit with a non-zero error code. However, in neither case will an error message be issued.

The files are read in the order given above, with last value found taking precedence over values read earlier. When multiple values are taken then all values of a key from all files will be used.

All writing options will per default write to the repository specific configuration file. Note that this also affects options like *--replace-all* and *--unset*. ***git config* will only ever change one file at a time**.

You can override these rules either by command-line options or by environment variables. The *--global* and the *--system* options will limit the file used to the global or system-wide file respectively. The GIT_CONFIG environment variable has a similar effect, but you can specify any filename you want.

ENVIRONMENT

GIT_CONFIG

Take the configuration from the given file instead of .git/config. Using the "--global" option forces this to ~/.gitconfig. Using the "--system" option forces this to $(prefix)/etc/gitconfig.

GIT_CONFIG_NOSYSTEM

Whether to skip reading settings from the system-wide $(prefix)/etc/gitconfig file. See git[1] for details.

See also FILES.

EXAMPLES

Given a .git/config like this:

```
#
# This is the config file, and
# a '#' or ';' character indicates
# a comment
#
```

```
; core variables
[core]
    ; Don't trust file modes
    filemode = false
```

```
; Our diff algorithm
[diff]
    external = /usr/local/bin/diff-wrapper
    renames = true
```

```
; Proxy settings
[core]
    gitproxy=proxy-command for kernel.org
    gitproxy=default-proxy ; for all the rest
```

```
; HTTP
[http]
    sslVerify
[http "https://weak.example.com"]
    sslVerify = false
    cookieFile = /tmp/cookie.txt
```

you can set the filemode to true with

```
% git config core.filemode true
```

The hypothetical proxy command entries actually have a postfix to discern what URL they apply to. Here is how to change the entry for kernel.org to "ssh".

```
% git config core.gitproxy '"ssh" for kernel.org' 'for kernel.org$'
```

This makes sure that only the key/value pair for kernel.org is replaced.

To delete the entry for renames, do

```
% git config --unset diff.renames
```

If you want to delete an entry for a multivar (like core.gitproxy above), you have to provide a regex matching the value of exactly one line.

To query the value for a given key, do

```
% git config --get core.filemode
```

or

```
% git config core.filemode
```

or, to query a multivar:

```
% git config --get core.gitproxy "for kernel.org$"
```

If you want to know all the values for a multivar, do:

```
% git config --get-all core.gitproxy
```

If you like to live dangerously, you can replace **all** core.gitproxy by a new one with

```
% git config --replace-all core.gitproxy ssh
```

However, if you really only want to replace the line for the default proxy, i.e. the one without a "for …" postfix, do something like this:

```
% git config core.gitproxy ssh '! for '
```

To actually match only values with an exclamation mark, you have to

```
% git config section.key value '[!]'
```

To add a new proxy, without altering any of the existing ones, use

```
% git config --add core.gitproxy '"proxy-command" for example.com'
```

An example to use customized color from the configuration in your script:

```
#!/bin/sh
WS=$(git config --get-color color.diff.whitespace "blue reverse")
RESET=$(git config --get-color "" "reset")
echo "${WS}your whitespace color or blue reverse${RESET}"
```

For URLs in `https://weak.example.com`, `http.sslVerify` is set to false, while it is set to `true` for all others:

```
% git config --bool --get-urlmatch http.sslverify https://good.example.com
true
% git config --bool --get-urlmatch http.sslverify https://weak.example.com
false
% git config --get-urlmatch http https://weak.example.com
http.cookieFile /tmp/cookie.txt
http.sslverify false
```

CONFIGURATION FILE

The Git configuration file contains a number of variables that affect the Git commands' behavior. The `.git/config` file in each repository is used to store the configuration for that repository, and `$HOME/.gitconfig` is used to store a per-user configuration as fallback values for the `.git/config` file. The file `/etc/gitconfig` can be used to store a system-wide default configuration.

The configuration variables are used by both the Git plumbing and the porcelains. The variables are divided into sections, wherein the fully qualified variable name of the variable itself is the last dot-separated segment and the section name is everything before the last dot. The variable names are case-insensitive, allow only alphanumeric characters and `-`, and must start with an alphabetic character. Some variables may appear multiple times; we say then that the variable is multivalued.

Syntax

The syntax is fairly flexible and permissive; whitespaces are mostly ignored. The # and *;* characters begin comments to the end of line, blank lines are ignored.

The file consists of sections and variables. A section begins with the name of the section in square brackets and continues until the next section begins. Section names are case-insensitive. Only alphanumeric characters, `-` and `.` are allowed in section names. Each variable must belong to some section, which means that there must be a section header before the first setting of a variable.

Sections can be further divided into subsections. To begin a subsection put its name in double quotes, separated by space from the section name, in the section header, like in the example below:

```
    [section "subsection"]
```

Subsection names are case sensitive and can contain any characters except newline (doublequote `"` and backslash can be included by escaping them as `\"` and `\\`, respectively). Section headers cannot span multiple lines. Variables may belong directly to a section or to a given subsection. You can have `[section]` if you have `[section "subsection"]`, but you don't need to.

There is also a deprecated `[section.subsection]` syntax. With this syntax, the subsection name is converted to lower-case and is also compared case sensitively. These subsection names follow the same restrictions as section names.

All the other lines (and the remainder of the line after the section header) are recognized as setting variables, in the form *name* = *value* (or just *name*, which is a short-hand to say that the variable is the boolean "true"). The variable names are case-insensitive, allow only alphanumeric characters and `-`, and must start with an alphabetic character.

A line that defines a value can be continued to the next line by ending it with a `\`; the backquote and the end-of-line are stripped. Leading whitespaces after *name* =, the remainder of the line after the first comment character # or *;*, and trailing whitespaces of the line are discarded unless they are enclosed in double quotes. Internal whitespaces within the value are retained verbatim.

Inside double quotes, double quote `"` and backslash `\` characters must be escaped: use `\"` for `"` and `\\` for `\`.

The following escape sequences (beside `\"` and `\\`) are recognized: `\n` for newline character (NL), `\t` for horizontal tabulation (HT, TAB) and `\b` for backspace (BS). Other char escape sequences (including octal escape sequences) are invalid.

Includes

You can include one config file from another by setting the special `include.path` variable to the name of the file to be included. The included file is expanded immediately, as if its contents had been found at the location of the include directive. If the value of the `include.path` variable is a relative path, the path is considered to be relative to the configuration file in which the include directive was found. The value of `include.path` is subject to tilde expansion: `~/` is expanded to the value of `$HOME`, and `~user/` to the specified user's home directory. See below for examples.

Example

```
# Core variables
[core]
    ; Don't trust file modes
    filemode = false
```

```
# Our diff algorithm
[diff]
    external = /usr/local/bin/diff-wrapper
    renames = true
```

```
[branch "devel"]
    remote = origin
    merge = refs/heads/devel
```

```
# Proxy settings
[core]
    gitProxy="ssh" for "kernel.org"
    gitProxy=default-proxy ; for the rest
```

```
[include]
    path = /path/to/foo.inc ; include by absolute path
    path = foo ; expand "foo" relative to the current file
    path = ~/foo ; expand "foo" in your $HOME directory
```

Values

Values of many variables are treated as a simple string, but there are variables that take values of specific types and there are rules as to how to spell them.

boolean

When a variable is said to take a boolean value, many synonyms are accepted for *true* and *false*; these are all case-insensitive.

true

Boolean true can be spelled as `yes` , `on` , `true` , or `1` . Also, a variable defined without `= <value>` is taken as true.

false

Boolean false can be spelled as `no` , `off` , `false` , or `0` .

When converting value to the canonical form using *--bool* type specifier; *git config* will ensure that the output is "true" or "false" (spelled in lowercase).

integer

The value for many variables that specify various sizes can be suffixed with `k`, `M`,... to mean "scale the number by 1024", "by 1024x1024", etc.

color

The value for a variables that takes a color is a list of colors (at most two) and attributes (at most one), separated by spaces. The colors accepted are `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` and `white`; the attributes are `bold`, `dim`, `ul`, `blink` and `reverse`. The first color given is the foreground; the second is the background. The position of the attribute, if any, doesn’t matter. Attributes may be turned off specifically by prefixing them with `no` (e.g., `noreverse`, `noul`, etc).

Colors (foreground and background) may also be given as numbers between 0 and 255; these use ANSI 256-color mode (but note that not all terminals may support this). If your terminal supports it, you may also specify 24-bit RGB values as hex, like `#ff0ab3`.

The attributes are meant to be reset at the beginning of each item in the colored output, so setting color.decorate.branch to `black` will paint that branch name in a plain `black`, even if the previous thing on the same output line (e.g. opening parenthesis before the list of branch names in `log --decorate` output) is set to be painted with `bold` or some other attribute.

Variables

Note that this list is non-comprehensive and not necessarily complete. For command-specific variables, you will find a more detailed description in the appropriate manual page.

Other git-related tools may and do use their own variables. When inventing new variables for use in your own tool, make sure their names do not conflict with those that are used by Git itself and other popular tools, and describe them in your documentation.

advice.*

These variables control various optional help messages designed to aid new users. All *advice.** variables default to *true*, and you can tell Git that you do not need help by setting these to *false*:

pushUpdateRejected

Set this variable to *false* if you want to disable *pushNonFFCurrent*, *pushNonFFMatching*, *pushAlreadyExists*, *pushFetchFirst*, and *pushNeedsForce* simultaneously.

pushNonFFCurrent

Advice shown when git-push[1] fails due to a non-fast-forward update to the current branch.

pushNonFFMatching

Advice shown when you ran [git-push[1]] and pushed *matching refs* explicitly (i.e. you used *:*, or specified a refspec that isn't your current branch) and it resulted in a non-fast-forward error.

pushAlreadyExists

Shown when git-push[1] rejects an update that does not qualify for fast-forwarding (e.g., a tag.)

pushFetchFirst

Shown when git-push[1] rejects an update that tries to overwrite a remote ref that points at an object we do not have.

pushNeedsForce

Shown when git-push[1] rejects an update that tries to overwrite a remote ref that points at an object that is not a commit-ish, or make the remote ref point at an object that is not a commit-ish.

statusHints

Show directions on how to proceed from the current state in the output of git-status[1], in the template shown when writing commit messages in git-commit[1], and in the help message shown by git-checkout[1] when switching branch.

statusUoption

Advise to consider using the `-u` option to git-status[1] when the command takes more than 2 seconds to enumerate untracked files.

commitBeforeMerge

Advice shown when git-merge[1] refuses to merge to avoid overwriting local changes.

resolveConflict

Advice shown by various commands when conflicts prevent the operation from being performed.

implicitIdentity

Advice on how to set your identity configuration when your information is guessed from the system username and domain name.

detachedHead

Advice shown when you used git-checkout[1] to move to the detach HEAD state, to instruct how to create a local branch after the fact.

amWorkDir

Advice that shows the location of the patch file when git-am[1] fails to apply it.

rmHints

In case of failure in the output of git-rm[1], show directions on how to proceed from the current state.

core.fileMode

Tells Git if the executable bit of files in the working tree is to be honored.

Some filesystems lose the executable bit when a file that is marked as executable is checked out, or checks out an non-executable file with executable bit on. git-clone[1] or git-init[1] probe the filesystem to see if it handles the executable bit correctly and this variable is automatically set as necessary.

A repository, however, may be on a filesystem that handles the filemode correctly, and this variable is set to *true* when created, but later may be made accessible from another environment that loses the filemode (e.g. exporting ext4 via CIFS mount, visiting a Cygwin created repository with Git for Windows or Eclipse). In such a case it may be necessary to set this variable to *false*. See git-update-index[1].

The default is true (when core.filemode is not specified in the config file).

core.ignoreCase

If true, this option enables various workarounds to enable Git to work better on filesystems that are not case sensitive, like FAT. For example, if a directory listing finds "makefile" when Git expects "Makefile", Git will assume it is really the same file, and continue to remember it as "Makefile".

The default is false, except git-clone[1] or git-init[1] will probe and set core.ignoreCase true if appropriate when the repository is created.

core.precomposeUnicode

This option is only used by Mac OS implementation of Git. When core.precomposeUnicode=true, Git reverts the unicode decomposition of filenames done by Mac OS. This is useful when sharing a repository between Mac OS and Linux or Windows. (Git for Windows 1.7.10 or higher is needed, or Git under cygwin 1.7). When false, file names are handled fully transparent by Git, which is backward compatible with older versions of Git.

core.protectHFS

If set to true, do not allow checkout of paths that would be considered equivalent to `.git` on an HFS+ filesystem. Defaults to `true` on Mac OS, and `false` elsewhere.

core.protectNTFS

If set to true, do not allow checkout of paths that would cause problems with the NTFS filesystem, e.g. conflict with 8.3 "short" names. Defaults to `true` on Windows, and `false` elsewhere.

core.trustctime

If false, the ctime differences between the index and the working tree are ignored; useful when the inode change time is regularly modified by something outside Git (file system crawlers and some backup systems). See git-update-index[1]. True by default.

core.untrackedCache

Determines what to do about the untracked cache feature of the index. It will be kept, if this variable is unset or set to `keep`. It will automatically be added if set to `true`. And it will automatically be removed, if set to `false`. Before setting it to `true`, you should check that mtime is working properly on your system. See git-update-index[1]. `keep` by default.

core.checkStat

Determines which stat fields to match between the index and work tree. The user can set this to *default* or *minimal*. Default (or explicitly *default*), is to check all fields, including the sub-second part of mtime and ctime.

core.quotePath

The commands that output paths (e.g. *ls-files*, *diff*), when not given the `-z` option, will quote "unusual" characters in the pathname by enclosing the pathname in a double-quote pair and with backslashes the same way strings in C source code are quoted. If this variable is set to false, the bytes higher than 0x80 are not quoted but output as verbatim. Note that double quote, backslash and control characters are always quoted without `-z` regardless of the setting of this variable.

core.eol

Sets the line ending type to use in the working directory for files that have the `text` property set. Alternatives are *lf*, *crlf* and *native*, which uses the platform's native line ending. The default value is `native`. See gitattributes[5] for more information on end-of-line conversion.

core.safecrlf

If true, makes Git check if converting `CRLF` is reversible when end-of-line conversion is active. Git will verify if a command modifies a file in the work tree either directly or indirectly. For example, committing a file followed by checking out the same file should yield the original file in the work tree. If this is not the case for the current setting of `core.autocrlf`, Git will reject the file. The variable can be set to "warn", in which case Git will only warn about an irreversible conversion but continue the operation.

CRLF conversion bears a slight chance of corrupting data. When it is enabled, Git will convert CRLF to LF during commit and LF to CRLF during checkout. A file that contains a mixture of LF and CRLF before the commit cannot be recreated by Git. For text files this is the right thing to do: it corrects line endings such that we have only LF line endings in the repository. But for binary files that are accidentally classified as text the conversion can corrupt data.

If you recognize such corruption early you can easily fix it by setting the conversion type explicitly in .gitattributes. Right after committing you still have the original file in your work tree and this file is not yet corrupted. You can explicitly tell Git that this file is binary and Git will handle the file appropriately.

Unfortunately, the desired effect of cleaning up text files with mixed line endings and the undesired effect of corrupting binary files cannot be distinguished. In both cases CRLFs are removed in an irreversible way. For text files this is the right thing to do because CRLFs are line endings, while for binary files converting CRLFs corrupts data.

Note, this safety check does not mean that a checkout will generate a file identical to the original file for a different setting of `core.eol` and `core.autocrlf`, but only for the current one. For example, a text file with `LF` would be accepted with `core.eol=lf` and could later be checked out with `core.eol=crlf`, in which case the resulting file would contain `CRLF`, although the original file contained `LF`. However, in both work trees the line endings would be consistent, that is either all `LF` or all `CRLF`, but never mixed. A file with mixed line endings would be reported by the `core.safecrlf` mechanism.

core.autocrlf

Setting this variable to "true" is almost the same as setting the `text` attribute to "auto" on all files except that text files are not guaranteed to be normalized: files that contain `CRLF` in the repository will not be touched. Use this setting if you want to have `CRLF` line endings in your working directory even though the repository does not have normalized line endings. This variable can be set to *input*, in which case no output conversion is performed.

core.symlinks

If false, symbolic links are checked out as small plain files that contain the link text. git-update-index[1] and git-add[1] will not change the recorded type to regular file. Useful on filesystems like FAT that do not support symbolic links.

The default is true, except git-clone[1] or git-init[1] will probe and set core.symlinks false if appropriate when the repository is created.

core.gitProxy

A "proxy command" to execute (as *command host port*) instead of establishing direct connection to the remote server when using the Git protocol for fetching. If the variable value is in the "COMMAND for DOMAIN" format, the command is applied only on hostnames ending with the specified domain string. This variable may be set multiple times and is matched in the given order; the first match wins.

Can be overridden by the *GIT_PROXY_COMMAND* environment variable (which always applies universally, without the special "for" handling).

The special string `none` can be used as the proxy command to specify that no proxy be used for a given domain pattern. This is useful for excluding servers inside a firewall from proxy use, while defaulting to a common proxy for external domains.

core.ignoreStat

If true, Git will avoid using lstat() calls to detect if files have changed by setting the "assume-unchanged" bit for those tracked files which it has updated identically in both the index and working tree.

When files are modified outside of Git, the user will need to stage the modified files explicitly (e.g. see *Examples* section in git-update-index[1]). Git will not normally detect changes to those files.

This is useful on systems where lstat() calls are very slow, such as CIFS/Microsoft Windows.

False by default.

core.preferSymlinkRefs

Instead of the default "symref" format for HEAD and other symbolic reference files, use symbolic links. This is sometimes needed to work with old scripts that expect HEAD to be a symbolic link.

core.bare

If true this repository is assumed to be *bare* and has no working directory associated with it. If this is the case a number of commands that require a working directory will be disabled, such as git-add[1] or git-merge[1].

This setting is automatically guessed by git-clone[1] or git-init[1] when the repository was created. By default a repository that ends in "/.git" is assumed to be not bare (bare = false), while all other repositories are assumed to be bare (bare = true).

core.worktree

Set the path to the root of the working tree. If GIT*COMMON_DIR environment variable is set, core.worktree is ignored and not used for determining the root of working tree. This can be overridden by the GIT_WORK_TREE environment variable and the* --work-tree_ command-line option. The value can be an absolute path or relative to the path to the .git directory, which is either specified by --git-dir or GIT_DIR, or automatically discovered. If --git-dir or GIT_DIR is specified but none of --work-tree, GIT_WORK_TREE and core.worktree is specified, the current working directory is regarded as the top level of your working tree.

Note that this variable is honored even when set in a configuration file in a ".git" subdirectory of a directory and its value differs from the latter directory (e.g. "/path/to/.git/config" has core.worktree set to "/different/path"), which is most likely a misconfiguration. Running Git commands in the "/path/to" directory will still use "/different/path" as the root of the work tree and can cause confusion unless you know what you are doing (e.g. you are creating a read-only snapshot of the same index to a location different from the repository's usual working tree).

core.logAllRefUpdates

Enable the reflog. Updates to a ref <ref> is logged to the file "$GIT_DIR/logs/<ref>", by appending the new and old SHA-1, the date/time and the reason of the update, but only when the file exists. If this configuration variable is set to true, missing "$GIT_DIR/logs/<ref>" file is automatically created for branch heads (i.e. under refs/heads/), remote refs (i.e. under refs/remotes/), note refs (i.e. under refs/notes/), and the symbolic ref HEAD.

This information can be used to determine what commit was the tip of a branch "2 days ago".

This value is true by default in a repository that has a working directory associated with it, and false by default in a bare repository.

core.repositoryFormatVersion

Internal variable identifying the repository format and layout version.

core.sharedRepository

When *group* (or *true*), the repository is made shareable between several users in a group (making sure all the files and objects are group-writable). When *all* (or *world* or *everybody*), the repository will be readable by all users, additionally to being group-shareable. When *umask* (or *false*), Git will use permissions reported by umask(2). When *0xxx*, where *0xxx* is an octal number, files in the repository will have this mode value. *0xxx* will override user's umask value (whereas the other options will only override requested parts of the user's umask value). Examples: *0660* will make the repo read/write-able for the owner and group, but inaccessible to others (equivalent to *group* unless umask is e.g. *0022*). *0640* is a repository that is group-readable but not group-writable. See git-init[1]. False by default.

core.warnAmbiguousRefs

If true, Git will warn you if the ref name you passed it is ambiguous and might match multiple refs in the repository. True by default.

core.compression

An integer -1..9, indicating a default compression level. -1 is the zlib default. 0 means no compression, and 1..9 are various speed/size tradeoffs, 9 being slowest. If set, this provides a default to other compression variables, such as *core.looseCompression* and *pack.compression*.

core.looseCompression

An integer -1..9, indicating the compression level for objects that are not in a pack file. -1 is the zlib default. 0 means no compression, and 1..9 are various speed/size tradeoffs, 9 being slowest. If not set, defaults to core.compression. If that is not set, defaults to 1 (best speed).

core.packedGitWindowSize

Number of bytes of a pack file to map into memory in a single mapping operation. Larger window sizes may allow your system to process a smaller number of large pack files more quickly. Smaller window sizes will negatively affect performance due to increased calls to the operating system's memory manager, but may improve performance when accessing a large number of large pack files.

Default is 1 MiB if NO_MMAP was set at compile time, otherwise 32 MiB on 32 bit platforms and 1 GiB on 64 bit platforms. This should be reasonable for all users/operating systems. You probably do not need to adjust this value.

Common unit suffixes of *k*, *m*, or *g* are supported.

core.packedGitLimit

Maximum number of bytes to map simultaneously into memory from pack files. If Git needs to access more than this many bytes at once to complete an operation it will unmap existing regions to reclaim virtual address space within the process.

Default is 256 MiB on 32 bit platforms and 8 GiB on 64 bit platforms. This should be reasonable for all users/operating systems, except on the largest projects. You probably do not need to adjust this value.

Common unit suffixes of *k*, *m*, or *g* are supported.

core.deltaBaseCacheLimit

Maximum number of bytes to reserve for caching base objects that may be referenced by multiple deltified objects. By storing the entire decompressed base objects in a cache Git is able to avoid unpacking and decompressing frequently used base objects multiple times.

Default is 96 MiB on all platforms. This should be reasonable for all users/operating systems, except on the largest projects. You probably do not need to adjust this value.

Common unit suffixes of *k*, *m*, or *g* are supported.

core.bigFileThreshold

Files larger than this size are stored deflated, without attempting delta compression. Storing large files without delta compression avoids excessive memory usage, at the slight expense of increased disk usage. Additionally files larger than this size are always treated as binary.

Default is 512 MiB on all platforms. This should be reasonable for most projects as source code and other text files can still be delta compressed, but larger binary media files won't be.

Common unit suffixes of *k*, *m*, or *g* are supported.

core.excludesFile

In addition to *.gitignore* (per-directory) and *.git/info/exclude*, Git looks into this file for patterns of files which are not meant to be tracked. " `~/` " is expanded to the value of `$HOME` and " `~user/` " to the specified user's home directory. Its default value is $XDG_CONFIG_HOME/git/ignore. If $XDG_CONFIG_HOME is either not set or empty, $HOME/.config/git/ignore is used instead. See gitignore[5].

core.askPass

Some commands (e.g. svn and http interfaces) that interactively ask for a password can be told to use an external program given via the value of this variable. Can be overridden by the *GIT_ASKPASS* environment variable. If not set, fall back to the value of the *SSH_ASKPASS*

environment variable or, failing that, a simple password prompt. The external program shall be given a suitable prompt as command-line argument and write the password on its STDOUT.

core.attributesFile

In addition to *.gitattributes* (per-directory) and *.git/info/attributes*, Git looks into this file for attributes (see gitattributes[5]). Path expansions are made the same way as for `core.excludesFile`. Its default value is $XDG_CONFIG_HOME/git/attributes. If $XDG_CONFIG_HOME is either not set or empty, $HOME/.config/git/attributes is used instead.

core.editor

Commands such as `commit` and `tag` that lets you edit messages by launching an editor uses the value of this variable when it is set, and the environment variable `GIT_EDITOR` is not set. See git-var[1].

core.commentChar

Commands such as `commit` and `tag` that lets you edit messages consider a line that begins with this character commented, and removes them after the editor returns (default #).

If set to "auto", `git-commit` would select a character that is not the beginning character of any line in existing commit messages.

core.packedRefsTimeout

The length of time, in milliseconds, to retry when trying to lock the `packed-refs` file. Value 0 means not to retry at all; -1 means to try indefinitely. Default is 1000 (i.e., retry for 1 second).

sequence.editor

Text editor used by `git rebase -i` for editing the rebase instruction file. The value is meant to be interpreted by the shell when it is used. It can be overridden by the `GIT_SEQUENCE_EDITOR` environment variable. When not configured the default commit message editor is used instead.

core.pager

Text viewer for use by Git commands (e.g., *less*). The value is meant to be interpreted by the shell. The order of preference is the `$GIT_PAGER` environment variable, then `core.pager` configuration, then `$PAGER`, and then the default chosen at compile time (usually *less*).

When the `LESS` environment variable is unset, Git sets it to `FRX` (if `LESS` environment variable is set, Git does not change it at all). If you want to selectively override Git's default setting for `LESS`, you can set `core.pager` to e.g. `less -S`. This will be passed to the shell

by Git, which will translate the final command to `LESS=FRX less -S`. The environment does not set the `S` option but the command line does, instructing less to truncate long lines. Similarly, setting `core.pager` to `less -+F` will deactivate the `F` option specified by the environment from the command-line, deactivating the "quit if one screen" behavior of `less`. One can specifically activate some flags for particular commands: for example, setting `pager.blame` to `less -S` enables line truncation only for `git blame`.

Likewise, when the `LV` environment variable is unset, Git sets it to `-c`. You can override this setting by exporting `LV` with another value or setting `core.pager` to `lv +c`.

core.whitespace

A comma separated list of common whitespace problems to notice. *git diff* will use `color.diff.whitespace` to highlight them, and *git apply --whitespace=error* will consider them as errors. You can prefix `-` to disable any of them (e.g. `-trailing-space`):

- `blank-at-eol` treats trailing whitespaces at the end of the line as an error (enabled by default).
- `space-before-tab` treats a space character that appears immediately before a tab character in the initial indent part of the line as an error (enabled by default).
- `indent-with-non-tab` treats a line that is indented with space characters instead of the equivalent tabs as an error (not enabled by default).
- `tab-in-indent` treats a tab character in the initial indent part of the line as an error (not enabled by default).
- `blank-at-eof` treats blank lines added at the end of file as an error (enabled by default).
- `trailing-space` is a short-hand to cover both `blank-at-eol` and `blank-at-eof`.
- `cr-at-eol` treats a carriage-return at the end of line as part of the line terminator, i.e. with it, `trailing-space` does not trigger if the character before such a carriage-return is not a whitespace (not enabled by default).
- `tabwidth=<n>` tells how many character positions a tab occupies; this is relevant for `indent-with-non-tab` and when Git fixes `tab-in-indent` errors. The default tab width is 8. Allowed values are 1 to 63.

core.fsyncObjectFiles

This boolean will enable *fsync()* when writing object files.

This is a total waste of time and effort on a filesystem that orders data writes properly, but can be useful for filesystems that do not use journalling (traditional UNIX filesystems) or that only journal metadata and not file contents (OS X's HFS+, or Linux ext3 with "data=writeback").

core.preloadIndex

Enable parallel index preload for operations like *git diff*

This can speed up operations like *git diff* and *git status* especially on filesystems like NFS that have weak caching semantics and thus relatively high IO latencies. When enabled, Git will do the index comparison to the filesystem data in parallel, allowing overlapping IO's. Defaults to true.

core.createObject

You can set this to *link*, in which case a hardlink followed by a delete of the source are used to make sure that object creation will not overwrite existing objects.

On some file system/operating system combinations, this is unreliable. Set this config setting to *rename* there; However, This will remove the check that makes sure that existing object files will not get overwritten.

core.notesRef

When showing commit messages, also show notes which are stored in the given ref. The ref must be fully qualified. If the given ref does not exist, it is not an error but means that no notes should be printed.

This setting defaults to "refs/notes/commits", and it can be overridden by the *GIT_NOTES_REF* environment variable. See git-notes[1].

core.sparseCheckout

Enable "sparse checkout" feature. See section "Sparse checkout" in git-read-tree[1] for more information.

core.abbrev

Set the length object names are abbreviated to. If unspecified, many commands abbreviate to 7 hexdigits, which may not be enough for abbreviated object names to stay unique for sufficiently long time.

add.ignoreErrors

add.ignore-errors (deprecated)

Tells *git add* to continue adding files when some files cannot be added due to indexing errors. Equivalent to the *--ignore-errors* option of git-add[1]. `add.ignore-errors` is deprecated, as it does not follow the usual naming convention for configuration variables.

alias.*

Command aliases for the git[1] command wrapper - e.g. after defining "alias.last = cat-file commit HEAD", the invocation "git last" is equivalent to "git cat-file commit HEAD". To avoid confusion and troubles with script usage, aliases that hide existing Git commands are ignored. Arguments are split by spaces, the usual shell quoting and escaping is supported. A quote pair or a backslash can be used to quote them.

If the alias expansion is prefixed with an exclamation point, it will be treated as a shell command. For example, defining "alias.new = !gitk --all --not ORIG*HEAD", the invocation "git new" is equivalent to running the shell command "gitk --all --not ORIG_HEAD". Note that shell commands will be executed from the top-level directory of a repository, which may not necessarily be the current directory. _GIT_PREFIX* is set as returned by running *git rev-parse --show-prefix* from the original current directory. See git-rev-parse[1].

am.keepcr

If true, git-am will call git-mailsplit for patches in mbox format with parameter *--keep-cr*. In this case git-mailsplit will not remove `\r` from lines ending with `\r\n`. Can be overridden by giving *--no-keep-cr* from the command line. See git-am[1], git-mailsplit[1].

am.threeWay

By default, `git am` will fail if the patch does not apply cleanly. When set to true, this setting tells `git am` to fall back on 3-way merge if the patch records the identity of blobs it is supposed to apply to and we have those blobs available locally (equivalent to giving the `--3way` option from the command line). Defaults to `false`. See git-am[1].

apply.ignoreWhitespace

When set to *change*, tells *git apply* to ignore changes in whitespace, in the same way as the *--ignore-space-change* option. When set to one of: no, none, never, false tells *git apply* to respect all whitespace differences. See git-apply[1].

apply.whitespace

Tells *git apply* how to handle whitespaces, in the same way as the *--whitespace* option. See git-apply[1].

branch.autoSetupMerge

Tells *git branch* and *git checkout* to set up new branches so that git-pull[1] will appropriately merge from the starting point branch. Note that even if this option is not set, this behavior can be chosen per-branch using the `--track` and `--no-track` options. The valid settings are: `false` — no automatic setup is done; `true` — automatic setup is done when the starting point is a remote-tracking branch; `always` — automatic setup is done when the starting point is either a local branch or remote-tracking branch. This option defaults to true.

branch.autoSetupRebase

When a new branch is created with *git branch* or *git checkout* that tracks another branch, this variable tells Git to set up pull to rebase instead of merge (see "branch.<name>.rebase"). When `never`, rebase is never automatically set to true. When `local`, rebase is set to true for tracked branches of other local branches. When `remote`, rebase is set to true for tracked branches of remote-tracking branches. When `always`, rebase will be set to true for all tracking branches. See "branch.autoSetupMerge" for details on how to set up a branch to track another branch. This option defaults to never.

branch.<name>.remote

When on branch <name>, it tells *git fetch* and *git push* which remote to fetch from/push to. The remote to push to may be overridden with `remote.pushDefault` (for all branches). The remote to push to, for the current branch, may be further overridden by `branch.<name>.pushRemote`. If no remote is configured, or if you are not on any branch, it defaults to `origin` for fetching and `remote.pushDefault` for pushing. Additionally, `.` (a period) is the current local repository (a dot-repository), see `branch.<name>.merge`'s final note below.

branch.<name>.pushRemote

When on branch <name>, it overrides `branch.<name>.remote` for pushing. It also overrides `remote.pushDefault` for pushing from branch <name>. When you pull from one place (e.g. your upstream) and push to another place (e.g. your own publishing repository), you would want to set `remote.pushDefault` to specify the remote to push to for all branches, and use this option to override it for a specific branch.

branch.<name>.merge

Defines, together with branch.<name>.remote, the upstream branch for the given branch. It tells *git fetch*/*git pull*/*git rebase* which branch to merge and can also affect *git push* (see push.default). When in branch <name>, it tells *git fetch* the default refspec to be marked for merging in FETCH*HEAD. The value is handled like the remote part of a refspec, and must match a ref which is fetched from the remote given by "branch.<name>.remote". The merge information is used by _git pull* (which at first calls *git fetch*) to lookup the default branch for merging. Without this option, *git pull* defaults to merge the first refspec fetched. Specify

multiple values to get an octopus merge. If you wish to setup *git pull* so that it merges into <name> from another branch in the local repository, you can point branch.<name>.merge to the desired branch, and use the relative path setting `.` (a period) for branch.<name>.remote.

branch.<name>.mergeOptions

Sets default options for merging into branch <name>. The syntax and supported options are the same as those of git-merge[1], but option values containing whitespace characters are currently not supported.

branch.<name>.rebase

When true, rebase the branch <name> on top of the fetched branch, instead of merging the default branch from the default remote when "git pull" is run. See "pull.rebase" for doing this in a non branch-specific manner.

When preserve, also pass `--preserve-merges` along to *git rebase* so that locally committed merge commits will not be flattened by running *git pull*.

When the value is `interactive`, the rebase is run in interactive mode.

NOTE: this is a possibly dangerous operation; do **not** use it unless you understand the implications (see git-rebase[1] for details).

branch.<name>.description

Branch description, can be edited with `git branch --edit-description`. Branch description is automatically added in the format-patch cover letter or request-pull summary.

browser.<tool>.cmd

Specify the command to invoke the specified browser. The specified command is evaluated in shell with the URLs passed as arguments. (See git-web{litdd}browse[1].)

browser.<tool>.path

Override the path for the given tool that may be used to browse HTML help (see *-w* option in git-help[1]) or a working repository in gitweb (see git-instaweb[1]).

clean.requireForce

A boolean to make git-clean do nothing unless given -f, -i or -n. Defaults to true.

color.branch

A boolean to enable/disable color in the output of git-branch[1]. May be set to `always`, `false` (or `never`) or `auto` (or `true`), in which case colors are used only when the output is to a terminal. Defaults to false.

color.branch.<slot>

Use customized color for branch coloration. `<slot>` is one of `current` (the current branch), `local` (a local branch), `remote` (a remote-tracking branch in refs/remotes/), `upstream` (upstream tracking branch), `plain` (other refs).

color.diff

Whether to use ANSI escape sequences to add color to patches. If this is set to `always`, git-diff[1], git-log[1], and git-show[1] will use color for all patches. If it is set to `true` or `auto`, those commands will only use color when output is to the terminal. Defaults to false.

This does not affect git-format-patch[1] or the *git-diff-** plumbing commands. Can be overridden on the command line with the `--color[=<when>]` option.

color.diff.<slot>

Use customized color for diff colorization. `<slot>` specifies which part of the patch to use the specified color, and is one of `context` (context text - `plain` is a historical synonym), `meta` (metainformation), `frag` (hunk header), *func* (function in hunk header), `old` (removed lines), `new` (added lines), `commit` (commit headers), or `whitespace` (highlighting whitespace errors).

color.decorate.<slot>

Use customized color for *git log --decorate* output. `<slot>` is one of `branch`, `remoteBranch`, `tag`, `stash` or `HEAD` for local branches, remote-tracking branches, tags, stash and HEAD, respectively.

color.grep

When set to `always`, always highlight matches. When `false` (or `never`), never. When set to `true` or `auto`, use color only when the output is written to the terminal. Defaults to `false`.

color.grep.<slot>

Use customized color for grep colorization. `<slot>` specifies which part of the line to use the specified color, and is one of

`context`

non-matching text in context lines (when using `-A`, `-B`, or `-C`)

`filename`

filename prefix (when not using `-h`)

`function`

function name lines (when using `-p`)

`linenumber`

line number prefix (when using `-n`)

`match`

matching text (same as setting `matchContext` and `matchSelected`)

`matchContext`

matching text in context lines

`matchSelected`

matching text in selected lines

`selected`

non-matching text in selected lines

`separator`

separators between fields on a line (`:`, `-`, and `=`) and between hunks (`--`)

color.interactive

When set to `always`, always use colors for interactive prompts and displays (such as those used by "git-add --interactive" and "git-clean --interactive"). When false (or `never`), never. When set to `true` or `auto`, use colors only when the output is to the terminal. Defaults to false.

color.interactive.<slot>

Use customized color for *git add --interactive* and *git clean --interactive* output. `<slot>` may be `prompt`, `header`, `help` or `error`, for four distinct types of normal output from interactive commands.

color.pager

A boolean to enable/disable colored output when the pager is in use (default is true).

color.showBranch

A boolean to enable/disable color in the output of git-show-branch[1]. May be set to `always`, `false` (or `never`) or `auto` (or `true`), in which case colors are used only when the output is to a terminal. Defaults to false.

color.status

A boolean to enable/disable color in the output of git-status[1]. May be set to `always`, `false` (or `never`) or `auto` (or `true`), in which case colors are used only when the output is to a terminal. Defaults to false.

color.status.<slot>

Use customized color for status colorization. `<slot>` is one of `header` (the header text of the status message), `added` or `updated` (files which are added but not committed), `changed` (files which are changed but not added in the index), `untracked` (files which are not tracked by Git), `branch` (the current branch), `nobranch` (the color the *no branch* warning is shown in, defaulting to red), or `unmerged` (files which have unmerged changes).

color.ui

This variable determines the default value for variables such as `color.diff` and `color.grep` that control the use of color per command family. Its scope will expand as more commands learn configuration to set a default for the `--color` option. Set it to `false` or `never` if you prefer Git commands not to use color unless enabled explicitly with some other configuration or the `--color` option. Set it to `always` if you want all output not intended for machine consumption to use color, to `true` or `auto` (this is the default since Git 1.8.4) if you want such output to use color when written to the terminal.

column.ui

Specify whether supported commands should output in columns. This variable consists of a list of tokens separated by spaces or commas:

These options control when the feature should be enabled (defaults to *never*):

`always`

always show in columns

`never`

never show in columns

`auto`

show in columns if the output is to the terminal

These options control layout (defaults to *column*). Setting any of these implies *always* if none of *always*, *never*, or *auto* are specified.

`column`

fill columns before rows

`row`

fill rows before columns

`plain`

show in one column

Finally, these options can be combined with a layout option (defaults to *nodense*):

`dense`

make unequal size columns to utilize more space

`nodense`

make equal size columns

column.branch

Specify whether to output branch listing in `git branch` in columns. See `column.ui` for details.

column.clean

Specify the layout when list items in `git clean -i`, which always shows files and directories in columns. See `column.ui` for details.

column.status

Specify whether to output untracked files in `git status` in columns. See `column.ui` for details.

column.tag

Specify whether to output tag listing in `git tag` in columns. See `column.ui` for details.

commit.cleanup

This setting overrides the default of the `--cleanup` option in `git commit`. See git-commit[1] for details. Changing the default can be useful when you always want to keep lines that begin with comment character `#` in your log message, in which case you would do `git config commit.cleanup whitespace` (note that you will have to remove the help lines that begin with `#` in the commit log template yourself, if you do this).

commit.gpgSign

A boolean to specify whether all commits should be GPG signed. Use of this option when doing operations such as rebase can result in a large number of commits being signed. It may be convenient to use an agent to avoid typing your GPG passphrase several times.

commit.status

A boolean to enable/disable inclusion of status information in the commit message template when using an editor to prepare the commit message. Defaults to true.

commit.template

Specify a file to use as the template for new commit messages. " `~/` " is expanded to the value of `$HOME` and " `~user/` " to the specified user's home directory.

credential.helper

Specify an external helper to be called when a username or password credential is needed; the helper may consult external storage to avoid prompting the user for the credentials. See gitcredentials[7] for details.

credential.useHttpPath

When acquiring credentials, consider the "path" component of an http or https URL to be important. Defaults to false. See gitcredentials[7] for more information.

credential.username

If no username is set for a network authentication, use this username by default. See credential.<context>.* below, and gitcredentials[7].

credential.<url>.*

Any of the credential.* options above can be applied selectively to some credentials. For example "credential.https://example.com.username" would set the default username only for https connections to example.com. See gitcredentials[7] for details on how URLs are matched.

credentialCache.ignoreSIGHUP

Tell git-credential-cache—daemon to ignore SIGHUP, instead of quitting.

diff-config.txt

difftool.<tool>.path

Override the path for the given tool. This is useful in case your tool is not in the PATH.

difftool.<tool>.cmd

Specify the command to invoke the specified diff tool. The specified command is evaluated in shell with the following variables available: *LOCAL* is set to the name of the temporary file containing the contents of the diff pre-image and *REMOTE* is set to the name of the temporary file containing the contents of the diff post-image.

difftool.prompt

Prompt before each invocation of the diff tool.

fetch.recurseSubmodules

This option can be either set to a boolean value or to *on-demand*. Setting it to a boolean changes the behavior of fetch and pull to unconditionally recurse into submodules when set to true or to not recurse at all when set to false. When set to *on-demand* (the default value), fetch and pull will only recurse into a populated submodule when its superproject retrieves a commit that updates the submodule’s reference.

fetch.fsckObjects

If it is set to true, git-fetch-pack will check all fetched objects. It will abort in the case of a malformed object or a broken link. The result of an abort are only dangling objects. Defaults to false. If not set, the value of `transfer.fsckObjects` is used instead.

fetch.unpackLimit

If the number of objects fetched over the Git native transfer is below this limit, then the objects will be unpacked into loose object files. However if the number of received objects equals or exceeds this limit then the received pack will be stored as a pack, after adding any missing delta bases. Storing the pack from a push can make the push operation complete faster, especially on slow filesystems. If not set, the value of `transfer.unpackLimit` is used instead.

fetch.prune

If true, fetch will automatically behave as if the `--prune` option was given on the command line. See also `remote.<name>.prune`.

format.attach

Enable multipart/mixed attachments as the default for *format-patch*. The value can also be a double quoted string which will enable attachments as the default and set the value as the boundary. See the --attach option in git-format-patch[1].

format.numbered

A boolean which can enable or disable sequence numbers in patch subjects. It defaults to "auto" which enables it only if there is more than one patch. It can be enabled or disabled for all messages by setting it to "true" or "false". See --numbered option in git-format-patch[1].

format.headers

Additional email headers to include in a patch to be submitted by mail. See git-format-patch[1].

format.to

format.cc

Additional recipients to include in a patch to be submitted by mail. See the --to and --cc options in git-format-patch[1].

format.subjectPrefix

The default for format-patch is to output files with the *[PATCH]* subject prefix. Use this variable to change that prefix.

format.signature

The default for format-patch is to output a signature containing the Git version number. Use this variable to change that default. Set this variable to the empty string ("") to suppress signature generation.

format.signatureFile

Works just like format.signature except the contents of the file specified by this variable will be used as the signature.

format.suffix

The default for format-patch is to output files with the suffix `.patch`. Use this variable to change that suffix (make sure to include the dot if you want it).

format.pretty

The default pretty format for log/show/whatchanged command, See git-log[1], git-show[1], git-whatchanged[1].

format.thread

The default threading style for *git format-patch*. Can be a boolean value, or `shallow` or `deep`. `shallow` threading makes every mail a reply to the head of the series, where the head is chosen from the cover letter, the `--in-reply-to`, and the first patch mail, in this

order. `deep` threading makes every mail a reply to the previous one. A true boolean value is the same as `shallow`, and a false value disables threading.

format.signOff

A boolean value which lets you enable the `-s/--signoff` option of format-patch by default. **Note:** Adding the Signed-off-by: line to a patch should be a conscious act and means that you certify you have the rights to submit this work under the same open source license. Please see the *SubmittingPatches* document for further discussion.

format.coverLetter

A boolean that controls whether to generate a cover-letter when format-patch is invoked, but in addition can be set to "auto", to generate a cover-letter only when there’s more than one patch.

format.outputDirectory

Set a custom directory to store the resulting files instead of the current working directory.

filter.<driver>.clean

The command which is used to convert the content of a worktree file to a blob upon checkin. See gitattributes[5] for details.

filter.<driver>.smudge

The command which is used to convert the content of a blob object to a worktree file upon checkout. See gitattributes[5] for details.

fsck.<msg-id>

Allows overriding the message type (error, warn or ignore) of a specific message ID such as `missingEmail`.

For convenience, fsck prefixes the error/warning with the message ID, e.g. "missingEmail: invalid author/committer line - missing email" means that setting `fsck.missingEmail = ignore` will hide that issue.

This feature is intended to support working with legacy repositories which cannot be repaired without disruptive changes.

fsck.skipList

The path to a sorted list of object names (i.e. one SHA-1 per line) that are known to be broken in a non-fatal way and should be ignored. This feature is useful when an established project should be accepted despite early commits containing errors that can be safely

ignored such as invalid committer email addresses. Note: corrupt objects cannot be skipped with this setting.

gc.aggressiveDepth

The depth parameter used in the delta compression algorithm used by *git gc --aggressive*. This defaults to 250.

gc.aggressiveWindow

The window size parameter used in the delta compression algorithm used by *git gc --aggressive*. This defaults to 250.

gc.auto

When there are approximately more than this many loose objects in the repository, `git gc --auto` will pack them. Some Porcelain commands use this command to perform a light-weight garbage collection from time to time. The default value is 6700. Setting this to 0 disables it.

gc.autoPackLimit

When there are more than this many packs that are not marked with `*.keep` file in the repository, `git gc --auto` consolidates them into one larger pack. The default value is 50. Setting this to 0 disables it.

gc.autoDetach

Make `git gc --auto` return immediately and run in background if the system supports it. Default is true.

gc.packRefs

Running `git pack-refs` in a repository renders it unclonable by Git versions prior to 1.5.1.2 over dumb transports such as HTTP. This variable determines whether *git gc* runs `git pack-refs`. This can be set to `notbare` to enable it within all non-bare repos or it can be set to a boolean value. The default is `true`.

gc.pruneExpire

When *git gc* is run, it will call *prune --expire 2.weeks.ago*. Override the grace period with this config variable. The value "now" may be used to disable this grace period and always prune unreachable objects immediately, or "never" may be used to suppress pruning.

gc.worktreePruneExpire

When *git gc* is run, it calls *git worktree prune --expire 3.months.ago*. This config variable can be used to set a different grace period. The value "now" may be used to disable the grace period and prune $GIT_DIR/worktrees immediately, or "never" may be used to suppress pruning.

gc.reflogExpire

gc.<pattern>.reflogExpire

git reflog expire removes reflog entries older than this time; defaults to 90 days. The value "now" expires all entries immediately, and "never" suppresses expiration altogether. With "<pattern>" (e.g. "refs/stash") in the middle the setting applies only to the refs that match the <pattern>.

gc.reflogExpireUnreachable

gc.<pattern>.reflogExpireUnreachable

git reflog expire removes reflog entries older than this time and are not reachable from the current tip; defaults to 30 days. The value "now" expires all entries immediately, and "never" suppresses expiration altogether. With "<pattern>" (e.g. "refs/stash") in the middle, the setting applies only to the refs that match the <pattern>.

gc.rerereResolved

Records of conflicted merge you resolved earlier are kept for this many days when *git rerere gc* is run. The default is 60 days. See git-rerere[1].

gc.rerereUnresolved

Records of conflicted merge you have not resolved are kept for this many days when *git rerere gc* is run. The default is 15 days. See git-rerere[1].

gitcvs.commitMsgAnnotation

Append this string to each commit message. Set to empty string to disable this feature. Defaults to "via git-CVS emulator".

gitcvs.enabled

Whether the CVS server interface is enabled for this repository. See git-cvsserver[1].

gitcvs.logFile

Path to a log file where the CVS server interface well… logs various stuff. See git-cvsserver[1].

gitcvs.usecrlfattr

If true, the server will look up the end-of-line conversion attributes for files to determine the -*k* modes to use. If the attributes force Git to treat a file as text, the *-k* mode will be left blank so CVS clients will treat it as text. If they suppress text conversion, the file will be set with *-kb* mode, which suppresses any newline munging the client might otherwise do. If the attributes do not allow the file type to be determined, then *gitcvs.allBinary* is used. See gitattributes[5].

gitcvs.allBinary

This is used if *gitcvs.usecrlfattr* does not resolve the correct *-kb* mode to use. If true, all unresolved files are sent to the client in mode *-kb*. This causes the client to treat them as binary files, which suppresses any newline munging it otherwise might do. Alternatively, if it is set to "guess", then the contents of the file are examined to decide if it is binary, similar to *core.autocrlf*.

gitcvs.dbName

Database used by git-cvsserver to cache revision information derived from the Git repository. The exact meaning depends on the used database driver, for SQLite (which is the default driver) this is a filename. Supports variable substitution (see git-cvsserver[1] for details). May not contain semicolons (`;`). Default: *%Ggitcvs.%m.sqlite*

gitcvs.dbDriver

Used Perl DBI driver. You can specify any available driver for this here, but it might not work. git-cvsserver is tested with *DBD::SQLite*, reported to work with *DBD::Pg*, and reported **not** to work with *DBD::mysql*. Experimental feature. May not contain double colons (`:`). Default: *SQLite*. See git-cvsserver[1].

gitcvs.dbUser, gitcvs.dbPass

Database user and password. Only useful if setting *gitcvs.dbDriver*, since SQLite has no concept of database users and/or passwords. *gitcvs.dbUser* supports variable substitution (see git-cvsserver[1] for details).

gitcvs.dbTableNamePrefix

Database table name prefix. Prepended to the names of any database tables used, allowing a single database to be used for several repositories. Supports variable substitution (see git-cvsserver[1] for details). Any non-alphabetic characters will be replaced with underscores.

All gitcvs variables except for *gitcvs.usecrlfattr* and *gitcvs.allBinary* can also be specified as *gitcvs.<access_method>.<varname>* (where *access_method* is one of "ext" and "pserver") to make them apply only for the given access method.

gitweb.category

gitweb.description

gitweb.owner

gitweb.url

See gitweb[1] for description.

gitweb.avatar

gitweb.blame

gitweb.grep

gitweb.highlight

gitweb.patches

gitweb.pickaxe

gitweb.remote_heads

gitweb.showSizes

gitweb.snapshot

See gitweb.conf[5] for description.

grep.lineNumber

If set to true, enable *-n* option by default.

grep.patternType

Set the default matching behavior. Using a value of *basic*, *extended*, *fixed*, or *perl* will enable the *--basic-regexp*, *--extended-regexp*, *--fixed-strings*, or *--perl-regexp* option accordingly, while the value *default* will return to the default matching behavior.

grep.extendedRegexp

If set to true, enable *--extended-regexp* option by default. This option is ignored when the *grep.patternType* option is set to a value other than *default*.

grep.threads

Number of grep worker threads to use. See `grep.threads` in git-grep[1] for more information.

grep.fallbackToNoIndex

If set to true, fall back to git grep --no-index if git grep is executed outside of a git repository. Defaults to false.

gpg.program

Use this custom program instead of "gpg" found on $PATH when making or verifying a PGP signature. The program must support the same command-line interface as GPG, namely, to verify a detached signature, "gpg --verify $file - <$signature" is run, and the program is expected to signal a good signature by exiting with code 0, and to generate an ASCII-armored detached signature, the standard input of "gpg -bsau $key" is fed with the contents to be signed, and the program is expected to send the result to its standard output.

gui.commitMsgWidth

Defines how wide the commit message window is in the git-gui[1]. "75" is the default.

gui.diffContext

Specifies how many context lines should be used in calls to diff made by the git-gui[1]. The default is "5".

gui.displayUntracked

Determines if :git-gui[1] shows untracked files in the file list. The default is "true".

gui.encoding

Specifies the default encoding to use for displaying of file contents in git-gui[1] and gitk[1]. It can be overridden by setting the *encoding* attribute for relevant files (see gitattributes[5]). If this option is not set, the tools default to the locale encoding.

gui.matchTrackingBranch

Determines if new branches created with git-gui[1] should default to tracking remote branches with matching names or not. Default: "false".

gui.newBranchTemplate

Is used as suggested name when creating new branches using the git-gui[1].

gui.pruneDuringFetch

"true" if git-gui[1] should prune remote-tracking branches when performing a fetch. The default value is "false".

gui.trustmtime

Determines if git-gui[1] should trust the file modification timestamp or not. By default the timestamps are not trusted.

gui.spellingDictionary

Specifies the dictionary used for spell checking commit messages in the git-gui[1]. When set to "none" spell checking is turned off.

gui.fastCopyBlame

If true, *git gui blame* uses `-C` instead of `-C -C` for original location detection. It makes blame significantly faster on huge repositories at the expense of less thorough copy detection.

gui.copyBlameThreshold

Specifies the threshold to use in *git gui blame* original location detection, measured in alphanumeric characters. See the git-blame[1] manual for more information on copy detection.

gui.blamehistoryctx

Specifies the radius of history context in days to show in gitk[1] for the selected commit, when the `Show History Context` menu item is invoked from *git gui blame*. If this variable is set to zero, the whole history is shown.

guitool.<name>.cmd

Specifies the shell command line to execute when the corresponding item of the git-gui[1] `Tools` menu is invoked. This option is mandatory for every tool. The command is executed from the root of the working directory, and in the environment it receives the name of the tool as *GIT_GUITOOL*, the name of the currently selected file as *FILENAME*, and the name of the current branch as *CUR_BRANCH* (if the head is detached, *CUR_BRANCH* is empty).

guitool.<name>.needsFile

Run the tool only if a diff is selected in the GUI. It guarantees that *FILENAME* is not empty.

guitool.<name>.noConsole

Run the command silently, without creating a window to display its output.

guitool.<name>.noRescan

Don’t rescan the working directory for changes after the tool finishes execution.

guitool.<name>.confirm

Show a confirmation dialog before actually running the tool.

guitool.<name>.argPrompt

Request a string argument from the user, and pass it to the tool through the *ARGS* environment variable. Since requesting an argument implies confirmation, the *confirm* option has no effect if this is enabled. If the option is set to *true*, *yes*, or *1*, the dialog uses a built-in generic prompt; otherwise the exact value of the variable is used.

guitool.<name>.revPrompt

Request a single valid revision from the user, and set the *REVISION* environment variable. In other aspects this option is similar to *argPrompt*, and can be used together with it.

guitool.<name>.revUnmerged

Show only unmerged branches in the *revPrompt* subdialog. This is useful for tools similar to merge or rebase, but not for things like checkout or reset.

guitool.<name>.title

Specifies the title to use for the prompt dialog. The default is the tool name.

guitool.<name>.prompt

Specifies the general prompt string to display at the top of the dialog, before subsections for *argPrompt* and *revPrompt*. The default value includes the actual command.

help.browser

Specify the browser that will be used to display help in the *web* format. See git-help[1].

help.format

Override the default help format used by git-help[1]. Values *man*, *info*, *web* and *html* are supported. *man* is the default. *web* and *html* are the same.

help.autoCorrect

Automatically correct and execute mistyped commands after waiting for the given number of deciseconds (0.1 sec). If more than one command can be deduced from the entered text, nothing will be executed. If the value of this option is negative, the corrected command will be executed immediately. If the value is 0 - the command will be just shown but not executed. This is the default.

help.htmlPath

Specify the path where the HTML documentation resides. File system paths and URLs are supported. HTML pages will be prefixed with this path when help is displayed in the *web* format. This defaults to the documentation path of your Git installation.

http.proxy

Override the HTTP proxy, normally configured using the *http_proxy*, *https_proxy*, and *all_proxy* environment variables (see `curl(1)`). In addition to the syntax understood by curl, it is possible to specify a proxy string with a user name but no password, in which case git will attempt to acquire one in the same way it does for other credentials. See gitcredentials[7] for more information. The syntax thus is *[protocol://][user[:password]@]proxyhost[:port]*. This can be overridden on a per-remote basis; see remote.<name>.proxy

http.proxyAuthMethod

Set the method with which to authenticate against the HTTP proxy. This only takes effect if the configured proxy string contains a user name part (i.e. is of the form *user@host* or *user@host:port*). This can be overridden on a per-remote basis; see `remote.<name>.proxyAuthMethod`. Both can be overridden by the *GIT_HTTP_PROXY_AUTHMETHOD* environment variable. Possible values are:

- `anyauth` - Automatically pick a suitable authentication method. It is assumed that the proxy answers an unauthenticated request with a 407 status code and one or more Proxy-authenticate headers with supported authentication methods. This is the default.
- `basic` - HTTP Basic authentication
- `digest` - HTTP Digest authentication; this prevents the password from being transmitted to the proxy in clear text
- `negotiate` - GSS-Negotiate authentication (compare the --negotiate option of `curl(1)`)
- `ntlm` - NTLM authentication (compare the --ntlm option of `curl(1)`)

http.emptyAuth

Attempt authentication without seeking a username or password. This can be used to attempt GSS-Negotiate authentication without specifying a username in the URL, as libcurl normally requires a username for authentication.

http.cookieFile

File containing previously stored cookie lines which should be used in the Git http session, if they match the server. The file format of the file to read cookies from should be plain HTTP headers or the Netscape/Mozilla cookie file format (see curl[1]). NOTE that the file specified with http.cookieFile is only used as input unless http.saveCookies is set.

http.saveCookies

If set, store cookies received during requests to the file specified by http.cookieFile. Has no effect if http.cookieFile is unset.

http.sslVersion

The SSL version to use when negotiating an SSL connection, if you want to force the default. The available and default version depend on whether libcurl was built against NSS or OpenSSL and the particular configuration of the crypto library in use. Internally this sets the *CURLOPT_SSL_VERSION* option; see the libcurl documentation for more details on the format of this option and for the ssl version supported. Actually the possible values of this option are:

- sslv2
- sslv3
- tlsv1
- tlsv1.0
- tlsv1.1
- tlsv1.2

Can be overridden by the *GIT_SSL_VERSION* environment variable. To force git to use libcurl's default ssl version and ignore any explicit http.sslversion option, set *GIT_SSL_VERSION* to the empty string.

http.sslCipherList

A list of SSL ciphers to use when negotiating an SSL connection. The available ciphers depend on whether libcurl was built against NSS or OpenSSL and the particular configuration of the crypto library in use. Internally this sets the *CURLOPT_SSL_CIPHER_LIST* option; see the libcurl documentation for more details on the format of this list.

Can be overridden by the *GIT_SSL_CIPHER_LIST* environment variable. To force git to use libcurl's default cipher list and ignore any explicit http.sslCipherList option, set *GIT_SSL_CIPHER_LIST* to the empty string.

http.sslVerify

Whether to verify the SSL certificate when fetching or pushing over HTTPS. Can be overridden by the *GIT_SSL_NO_VERIFY* environment variable.

http.sslCert

File containing the SSL certificate when fetching or pushing over HTTPS. Can be overridden by the *GIT_SSL_CERT* environment variable.

http.sslKey

File containing the SSL private key when fetching or pushing over HTTPS. Can be overridden by the *GIT_SSL_KEY* environment variable.

http.sslCertPasswordProtected

Enable Git's password prompt for the SSL certificate. Otherwise OpenSSL will prompt the user, possibly many times, if the certificate or private key is encrypted. Can be overridden by the *GIT_SSL_CERT_PASSWORD_PROTECTED* environment variable.

http.sslCAInfo

File containing the certificates to verify the peer with when fetching or pushing over HTTPS. Can be overridden by the *GIT_SSL_CAINFO* environment variable.

http.sslCAPath

Path containing files with the CA certificates to verify the peer with when fetching or pushing over HTTPS. Can be overridden by the *GIT_SSL_CAPATH* environment variable.

http.pinnedpubkey

Public key of the https service. It may either be the filename of a PEM or DER encoded public key file or a string starting with *sha256//* followed by the base64 encoded sha256 hash of the public key. See also libcurl *CURLOPT_PINNEDPUBLICKEY*. git will exit with an error if this option is set but not supported by cURL.

http.sslTry

Attempt to use AUTH SSL/TLS and encrypted data transfers when connecting via regular FTP protocol. This might be needed if the FTP server requires it for security reasons or you wish to connect securely whenever remote FTP server supports it. Default is false since it might trigger certificate verification errors on misconfigured servers.

http.maxRequests

How many HTTP requests to launch in parallel. Can be overridden by the *GIT_HTTP_MAX_REQUESTS* environment variable. Default is 5.

http.minSessions

The number of curl sessions (counted across slots) to be kept across requests. They will not be ended with curl_easy_cleanup() until http_cleanup() is invoked. If USE_CURL_MULTI is not defined, this value will be capped at 1. Defaults to 1.

http.postBuffer

Maximum size in bytes of the buffer used by smart HTTP transports when POSTing data to the remote system. For requests larger than this buffer size, HTTP/1.1 and Transfer-Encoding: chunked is used to avoid creating a massive pack file locally. Default is 1 MiB, which is sufficient for most requests.

http.lowSpeedLimit, http.lowSpeedTime

If the HTTP transfer speed is less than *http.lowSpeedLimit* for longer than *http.lowSpeedTime* seconds, the transfer is aborted. Can be overridden by the *GIT_HTTP_LOW_SPEED_LIMIT* and *GIT_HTTP_LOW_SPEED_TIME* environment variables.

http.noEPSV

A boolean which disables using of EPSV ftp command by curl. This can helpful with some "poor" ftp servers which don’t support EPSV mode. Can be overridden by the *GIT_CURL_FTP_NO_EPSV* environment variable. Default is false (curl will use EPSV).

http.userAgent

The HTTP USER*AGENT string presented to an HTTP server. The default value represents the version of the client Git such as git/1.7.1. This option allows you to override this value to a more common value such as Mozilla/4.0. This may be necessary, for instance, if connecting through a firewall that restricts HTTP connections to a set of common USER_AGENT strings (but not including those like git/1.7.1). Can be overridden by the _GIT_HTTP_USER_AGENT* environment variable.

http.<url>.*

Any of the http.* options above can be applied selectively to some URLs. For a config key to match a URL, each element of the config key is compared to that of the URL, in the following order:

1. Scheme (e.g., `https` in `https://example.com/`). This field must match exactly between the config key and the URL.

2. Host/domain name (e.g., `example.com` in `https://example.com/`). This field must match exactly between the config key and the URL.

3. Port number (e.g., `8080` in `http://example.com:8080/`). This field must match exactly between the config key and the URL. Omitted port numbers are automatically converted to the correct default for the scheme before matching.

4. Path (e.g., `repo.git` in `https://example.com/repo.git`). The path field of the config key must match the path field of the URL either exactly or as a prefix of slash-delimited path elements. This means a config key with path `foo/` matches URL path `foo/bar`. A prefix can only match on a slash (`/`) boundary. Longer matches take precedence (so a config key with path `foo/bar` is a better match to URL path `foo/bar` than a config key with just path `foo/`).

5. User name (e.g., `user` in `https://user@example.com/repo.git`). If the config key has a user name it must match the user name in the URL exactly. If the config key does not have a user name, that config key will match a URL with any user name (including none), but at a lower precedence than a config key with a user name.

The list above is ordered by decreasing precedence; a URL that matches a config key's path is preferred to one that matches its user name. For example, if the URL is `https://user@example.com/foo/bar` a config key match of `https://example.com/foo` will be preferred over a config key match of `https://user@example.com`.

All URLs are normalized before attempting any matching (the password part, if embedded in the URL, is always ignored for matching purposes) so that equivalent URLs that are simply spelled differently will match properly. Environment variable settings always override any matches. The URLs that are matched against are those given directly to Git commands. This means any URLs visited as a result of a redirection do not participate in matching.

i18n.commitEncoding

Character encoding the commit messages are stored in; Git itself does not care per se, but this information is necessary e.g. when importing commits from emails or in the gitk graphical history browser (and possibly at other places in the future or in other porcelains). See e.g. git-mailinfo[1]. Defaults to *utf-8*.

i18n.logOutputEncoding

Character encoding the commit messages are converted to when running *git log* and friends.

imap

The configuration variables in the *imap* section are described in git-imap-send[1].

index.version

Specify the version with which new index files should be initialized. This does not affect existing repositories.

init.templateDir

Specify the directory from which templates will be copied. (See the "TEMPLATE DIRECTORY" section of git-init[1].)

instaweb.browser

Specify the program that will be used to browse your working repository in gitweb. See git-instaweb[1].

instaweb.httpd

The HTTP daemon command-line to start gitweb on your working repository. See git-instaweb[1].

instaweb.local

If true the web server started by git-instaweb[1] will be bound to the local IP (127.0.0.1).

instaweb.modulePath

The default module path for git-instaweb[1] to use instead of /usr/lib/apache2/modules. Only used if httpd is Apache.

instaweb.port

The port number to bind the gitweb httpd to. See git-instaweb[1].

interactive.singleKey

In interactive commands, allow the user to provide one-letter input with a single key (i.e., without hitting enter). Currently this is used by the `--patch` mode of git-add[1], git-checkout[1], git-commit[1], git-reset[1], and git-stash[1]. Note that this setting is silently ignored if portable keystroke input is not available; requires the Perl module Term::ReadKey.

log.abbrevCommit

If true, makes git-log[1], git-show[1], and git-whatchanged[1] assume `--abbrev-commit`. You may override this option with `--no-abbrev-commit`.

log.date

Set the default date-time mode for the *log* command. Setting a value for log.date is similar to using *git log*'s `--date` option. See git-log[1] for details.

log.decorate

Print out the ref names of any commits that are shown by the log command. If *short* is specified, the ref name prefixes *refs/heads/*, *refs/tags/* and *refs/remotes/* will not be printed. If *full* is specified, the full ref name (including prefix) will be printed. This is the same as the log commands *--decorate* option.

log.follow

If `true`, `git log` will act as if the `--follow` option was used when a single <path> is given. This has the same limitations as `--follow`, i.e. it cannot be used to follow multiple files and does not work well on non-linear history.

log.showRoot

If true, the initial commit will be shown as a big creation event. This is equivalent to a diff against an empty tree. Tools like git-log[1] or git-whatchanged[1], which normally hide the root commit will now show it. True by default.

log.mailmap

If true, makes git-log[1], git-show[1], and git-whatchanged[1] assume `--use-mailmap`.

mailinfo.scissors

If true, makes git-mailinfo[1] (and therefore git-am[1]) act by default as if the --scissors option was provided on the command-line. When active, this features removes everything from the message body before a scissors line (i.e. consisting mainly of ">8", "8<" and "-").

mailmap.file

The location of an augmenting mailmap file. The default mailmap, located in the root of the repository, is loaded first, then the mailmap file pointed to by this variable. The location of the mailmap file may be in a repository subdirectory, or somewhere outside of the repository itself. See git-shortlog[1] and git-blame[1].

mailmap.blob

Like `mailmap.file`, but consider the value as a reference to a blob in the repository. If both `mailmap.file` and `mailmap.blob` are given, both are parsed, with entries from `mailmap.file` taking precedence. In a bare repository, this defaults to `HEAD:.mailmap`. In a non-bare repository, it defaults to empty.

man.viewer

Specify the programs that may be used to display help in the *man* format. See git-help[1].

man.<tool>.cmd

Specify the command to invoke the specified man viewer. The specified command is evaluated in shell with the man page passed as argument. (See git-help[1].)

man.<tool>.path

Override the path for the given tool that may be used to display help in the *man* format. See git-help[1].

merge-config.txt

mergetool.<tool>.path

Override the path for the given tool. This is useful in case your tool is not in the PATH.

mergetool.<tool>.cmd

Specify the command to invoke the specified merge tool. The specified command is evaluated in shell with the following variables available: *BASE* is the name of a temporary file containing the common base of the files to be merged, if available; *LOCAL* is the name of a temporary file containing the contents of the file on the current branch; *REMOTE* is the name of a temporary file containing the contents of the file from the branch being merged; *MERGED* contains the name of the file to which the merge tool should write the results of a successful merge.

mergetool.<tool>.trustExitCode

For a custom merge command, specify whether the exit code of the merge command can be used to determine whether the merge was successful. If this is not set to true then the merge target file timestamp is checked and the merge assumed to have been successful if the file has been updated, otherwise the user is prompted to indicate the success of the merge.

mergetool.meld.hasOutput

Older versions of `meld` do not support the `--output` option. Git will attempt to detect whether `meld` supports `--output` by inspecting the output of `meld --help`. Configuring `mergetool.meld.hasOutput` will make Git skip these checks and use the configured value instead. Setting `mergetool.meld.hasOutput` to `true` tells Git to unconditionally use the `--output` option, and `false` avoids using `--output`.

mergetool.keepBackup

After performing a merge, the original file with conflict markers can be saved as a file with a `.orig` extension. If this variable is set to `false` then this file is not preserved. Defaults to `true` (i.e. keep the backup files).

mergetool.keepTemporaries

When invoking a custom merge tool, Git uses a set of temporary files to pass to the tool. If the tool returns an error and this variable is set to `true`, then these temporary files will be preserved, otherwise they will be removed after the tool has exited. Defaults to `false`.

mergetool.writeToTemp

Git writes temporary *BASE*, *LOCAL*, and *REMOTE* versions of conflicting files in the worktree by default. Git will attempt to use a temporary directory for these files when set `true`. Defaults to `false`.

mergetool.prompt

Prompt before each invocation of the merge resolution program.

notes.mergeStrategy

Which merge strategy to choose by default when resolving notes conflicts. Must be one of `manual`, `ours`, `theirs`, `union`, or `cat_sort_uniq`. Defaults to `manual`. See "NOTES MERGE STRATEGIES" section of git-notes[1] for more information on each strategy.

notes.<name>.mergeStrategy

Which merge strategy to choose when doing a notes merge into refs/notes/<name>. This overrides the more general "notes.mergeStrategy". See the "NOTES MERGE STRATEGIES" section in git-notes[1] for more information on the available strategies.

notes.displayRef

The (fully qualified) refname from which to show notes when showing commit messages. The value of this variable can be set to a glob, in which case notes from all matching refs will be shown. You may also specify this configuration variable several times. A warning will be issued for refs that do not exist, but a glob that does not match any refs is silently ignored.

This setting can be overridden with the `GIT_NOTES_DISPLAY_REF` environment variable, which must be a colon separated list of refs or globs.

The effective value of "core.notesRef" (possibly overridden by GIT_NOTES_REF) is also implicitly added to the list of refs to be displayed.

notes.rewrite.<command>

When rewriting commits with <command> (currently `amend` or `rebase`) and this variable is set to `true`, Git automatically copies your notes from the original to the rewritten commit. Defaults to `true`, but see "notes.rewriteRef" below.

notes.rewriteMode

When copying notes during a rewrite (see the "notes.rewrite.<command>" option), determines what to do if the target commit already has a note. Must be one of `overwrite`, `concatenate`, `cat_sort_uniq`, or `ignore`. Defaults to `concatenate`.

This setting can be overridden with the `GIT_NOTES_REWRITE_MODE` environment variable.

notes.rewriteRef

When copying notes during a rewrite, specifies the (fully qualified) ref whose notes should be copied. The ref may be a glob, in which case notes in all matching refs will be copied. You may also specify this configuration several times.

Does not have a default value; you must configure this variable to enable note rewriting. Set it to `refs/notes/commits` to enable rewriting for the default commit notes.

This setting can be overridden with the `GIT_NOTES_REWRITE_REF` environment variable, which must be a colon separated list of refs or globs.

pack.window

The size of the window used by git-pack-objects[1] when no window size is given on the command line. Defaults to 10.

pack.depth

The maximum delta depth used by git-pack-objects[1] when no maximum depth is given on the command line. Defaults to 50.

pack.windowMemory

The maximum size of memory that is consumed by each thread in git-pack-objects[1] for pack window memory when no limit is given on the command line. The value can be suffixed with "k", "m", or "g". When left unconfigured (or set explicitly to 0), there will be no limit.

pack.compression

An integer -1..9, indicating the compression level for objects in a pack file. -1 is the zlib default. 0 means no compression, and 1..9 are various speed/size tradeoffs, 9 being slowest. If not set, defaults to core.compression. If that is not set, defaults to -1, the zlib default, which is "a default compromise between speed and compression (currently equivalent to level 6)."

Note that changing the compression level will not automatically recompress all existing objects. You can force recompression by passing the -F option to git-repack[1].

pack.deltaCacheSize

The maximum memory in bytes used for caching deltas in git-pack-objects[1] before writing them out to a pack. This cache is used to speed up the writing object phase by not having to recompute the final delta result once the best match for all objects is found. Repacking large repositories on machines which are tight with memory might be badly impacted by this though, especially if this cache pushes the system into swapping. A value of 0 means no limit. The smallest size of 1 byte may be used to virtually disable this cache. Defaults to 256 MiB.

pack.deltaCacheLimit

The maximum size of a delta, that is cached in git-pack-objects[1]. This cache is used to speed up the writing object phase by not having to recompute the final delta result once the best match for all objects is found. Defaults to 1000.

pack.threads

Specifies the number of threads to spawn when searching for best delta matches. This requires that git-pack-objects[1] be compiled with pthreads otherwise this option is ignored with a warning. This is meant to reduce packing time on multiprocessor machines. The required amount of memory for the delta search window is however multiplied by the number of threads. Specifying 0 will cause Git to auto-detect the number of CPU's and set the number of threads accordingly.

pack.indexVersion

Specify the default pack index version. Valid values are 1 for legacy pack index used by Git versions prior to 1.5.2, and 2 for the new pack index with capabilities for packs larger than 4 GB as well as proper protection against the repacking of corrupted packs. Version 2 is the default. Note that version 2 is enforced and this config option ignored whenever the corresponding pack is larger than 2 GB.

If you have an old Git that does not understand the version 2 `*.idx` file, cloning or fetching over a non native protocol (e.g. "http") that will copy both `*.pack` file and corresponding `*.idx` file from the other side may give you a repository that cannot be accessed with your older version of Git. If the `*.pack` file is smaller than 2 GB, however, you can use git-index-pack[1] on the *.pack file to regenerate the* `.idx` file.

pack.packSizeLimit

The maximum size of a pack. This setting only affects packing to a file when repacking, i.e. the git:// protocol is unaffected. It can be overridden by the `--max-pack-size` option of git-repack[1]. The minimum size allowed is limited to 1 MiB. The default is unlimited. Common unit suffixes of *k*, *m*, or *g* are supported.

pack.useBitmaps

When true, git will use pack bitmaps (if available) when packing to stdout (e.g., during the server side of a fetch). Defaults to true. You should not generally need to turn this off unless you are debugging pack bitmaps.

pack.writeBitmaps (deprecated)

This is a deprecated synonym for `repack.writeBitmaps`.

pack.writeBitmapHashCache

When true, git will include a "hash cache" section in the bitmap index (if one is written). This cache can be used to feed git’s delta heuristics, potentially leading to better deltas between bitmapped and non-bitmapped objects (e.g., when serving a fetch between an older, bitmapped pack and objects that have been pushed since the last gc). The downside is that it consumes 4 bytes per object of disk space, and that JGit’s bitmap implementation does not understand it, causing it to complain if Git and JGit are used on the same repository. Defaults to false.

pager.<cmd>

If the value is boolean, turns on or off pagination of the output of a particular Git subcommand when writing to a tty. Otherwise, turns on pagination for the subcommand using the pager specified by the value of `pager.<cmd>`. If `--paginate` or `--no-pager` is specified on the command line, it takes precedence over this option. To disable pagination for all commands, set `core.pager` or `GIT_PAGER` to `cat`.

pretty.<name>

Alias for a --pretty= format string, as specified in git-log[1]. Any aliases defined here can be used just as the built-in pretty formats could. For example, running `git config pretty.changelog "format:* %H %s"` would cause the invocation `git log --pretty=changelog` to be equivalent to running `git log "--pretty=format:* %H %s"`. Note that an alias with the same name as a built-in format will be silently ignored.

pull.ff

By default, Git does not create an extra merge commit when merging a commit that is a descendant of the current commit. Instead, the tip of the current branch is fast-forwarded. When set to `false`, this variable tells Git to create an extra merge commit in such a case (equivalent to giving the `--no-ff` option from the command line). When set to `only`, only such fast-forward merges are allowed (equivalent to giving the `--ff-only` option from the command line). This setting overrides `merge.ff` when pulling.

pull.rebase

When true, rebase branches on top of the fetched branch, instead of merging the default branch from the default remote when "git pull" is run. See "branch.<name>.rebase" for setting this on a per-branch basis.

When preserve, also pass `--preserve-merges` along to *git rebase* so that locally committed merge commits will not be flattened by running *git pull*.

When the value is `interactive`, the rebase is run in interactive mode.

NOTE: this is a possibly dangerous operation; do **not** use it unless you understand the implications (see git-rebase[1] for details).

pull.octopus

The default merge strategy to use when pulling multiple branches at once.

pull.twohead

The default merge strategy to use when pulling a single branch.

push.default

Defines the action `git push` should take if no refspec is explicitly given. Different values are well-suited for specific workflows; for instance, in a purely central workflow (i.e. the fetch source is equal to the push destination), `upstream` is probably what you want. Possible values are:

- `nothing` - do not push anything (error out) unless a refspec is explicitly given. This is primarily meant for people who want to avoid mistakes by always being explicit.
- `current` - push the current branch to update a branch with the same name on the receiving end. Works in both central and non-central workflows.
- `upstream` - push the current branch back to the branch whose changes are usually integrated into the current branch (which is called `@{upstream}`). This mode only makes sense if you are pushing to the same repository you would normally pull from (i.e. central workflow).
- `simple` - in centralized workflow, work like `upstream` with an added safety to refuse to push if the upstream branch's name is different from the local one.

  When pushing to a remote that is different from the remote you normally pull from, work as `current`. This is the safest option and is suited for beginners.

  This mode has become the default in Git 2.0.
- `matching` - push all branches having the same name on both ends. This makes the repository you are pushing to remember the set of branches that will be pushed out (e.g. if you always push *maint* and *master* there and no other branches, the repository you push to will have these two branches, and your local *maint* and *master* will be pushed there).

  To use this mode effectively, you have to make sure *all* the branches you would push out are ready to be pushed out before running *git push*, as the whole point of this mode is to allow you to push all of the branches in one go. If you usually finish work on only one branch and push out the result, while other branches are unfinished, this mode is

not for you. Also this mode is not suitable for pushing into a shared central repository, as other people may add new branches there, or update the tip of existing branches outside your control.

This used to be the default, but not since Git 2.0 (`simple` is the new default).

push.followTags

If set to true enable *--follow-tags* option by default. You may override this configuration at time of push by specifying *--no-follow-tags*.

push.gpgSign

May be set to a boolean value, or the string *if-asked*. A true value causes all pushes to be GPG signed, as if *--signed* is passed to git-push[1]. The string *if-asked* causes pushes to be signed if the server supports it, as if *--signed=if-asked* is passed to *git push*. A false value may override a value from a lower-priority config file. An explicit command-line flag always overrides this config option.

push.recurseSubmodules

Make sure all submodule commits used by the revisions to be pushed are available on a remote-tracking branch. If the value is *check* then Git will verify that all submodule commits that changed in the revisions to be pushed are available on at least one remote of the submodule. If any commits are missing, the push will be aborted and exit with non-zero status. If the value is *on-demand* then all submodules that changed in the revisions to be pushed will be pushed. If on-demand was not able to push all necessary revisions it will also be aborted and exit with non-zero status. If the value is *no* then default behavior of ignoring submodules when pushing is retained. You may override this configuration at time of push by specifying *--recurse-submodules=check|on-demand|no*.

rebase.stat

Whether to show a diffstat of what changed upstream since the last rebase. False by default.

rebase.autoSquash

If set to true enable *--autosquash* option by default.

rebase.autoStash

When set to true, automatically create a temporary stash before the operation begins, and apply it after the operation ends. This means that you can run rebase on a dirty worktree. However, use with care: the final stash application after a successful rebase might result in non-trivial conflicts. Defaults to false.

rebase.missingCommitsCheck

If set to "warn", git rebase -i will print a warning if some commits are removed (e.g. a line was deleted), however the rebase will still proceed. If set to "error", it will print the previous warning and stop the rebase, *git rebase --edit-todo* can then be used to correct the error. If set to "ignore", no checking is done. To drop a commit without warning or error, use the `drop` command in the todo-list. Defaults to "ignore".

rebase.instructionFormat A format string, as specified in git-log[1], to be used for the instruction list during an interactive rebase. The format will automatically have the long commit hash prepended to the format.

receive.advertiseAtomic

By default, git-receive-pack will advertise the atomic push capability to its clients. If you don’t want to this capability to be advertised, set this variable to false.

receive.autogc

By default, git-receive-pack will run "git-gc --auto" after receiving data from git-push and updating refs. You can stop it by setting this variable to false.

receive.certNonceSeed

By setting this variable to a string, `git receive-pack` will accept a `git push --signed` and verifies it by using a "nonce" protected by HMAC using this string as a secret key.

receive.certNonceSlop

When a `git push --signed` sent a push certificate with a "nonce" that was issued by a receive-pack serving the same repository within this many seconds, export the "nonce" found in the certificate to `GIT_PUSH_CERT_NONCE` to the hooks (instead of what the receive-pack asked the sending side to include). This may allow writing checks in `pre-receive` and `post-receive` a bit easier. Instead of checking `GIT_PUSH_CERT_NONCE_SLOP` environment variable that records by how many seconds the nonce is stale to decide if they want to accept the certificate, they only can check `GIT_PUSH_CERT_NONCE_STATUS` is `OK` .

receive.fsckObjects

If it is set to true, git-receive-pack will check all received objects. It will abort in the case of a malformed object or a broken link. The result of an abort are only dangling objects. Defaults to false. If not set, the value of `transfer.fsckObjects` is used instead.

receive.fsck.<msg-id>

When `receive.fsckObjects` is set to true, errors can be switched to warnings and vice versa by configuring the `receive.fsck.<msg-id>` setting where the `<msg-id>` is the fsck message ID and the value is one of `error` , `warn` or `ignore` . For convenience, fsck

prefixes the error/warning with the message ID, e.g. "missingEmail: invalid author/committer line - missing email" means that setting `receive.fsck.missingEmail = ignore` will hide that issue.

This feature is intended to support working with legacy repositories which would not pass pushing when `receive.fsckObjects = true`, allowing the host to accept repositories with certain known issues but still catch other issues.

receive.fsck.skipList

The path to a sorted list of object names (i.e. one SHA-1 per line) that are known to be broken in a non-fatal way and should be ignored. This feature is useful when an established project should be accepted despite early commits containing errors that can be safely ignored such as invalid committer email addresses. Note: corrupt objects cannot be skipped with this setting.

receive.unpackLimit

If the number of objects received in a push is below this limit then the objects will be unpacked into loose object files. However if the number of received objects equals or exceeds this limit then the received pack will be stored as a pack, after adding any missing delta bases. Storing the pack from a push can make the push operation complete faster, especially on slow filesystems. If not set, the value of `transfer.unpackLimit` is used instead.

receive.denyDeletes

If set to true, git-receive-pack will deny a ref update that deletes the ref. Use this to prevent such a ref deletion via a push.

receive.denyDeleteCurrent

If set to true, git-receive-pack will deny a ref update that deletes the currently checked out branch of a non-bare repository.

receive.denyCurrentBranch

If set to true or "refuse", git-receive-pack will deny a ref update to the currently checked out branch of a non-bare repository. Such a push is potentially dangerous because it brings the HEAD out of sync with the index and working tree. If set to "warn", print a warning of such a push to stderr, but allow the push to proceed. If set to false or "ignore", allow such pushes with no message. Defaults to "refuse".

Another option is "updateInstead" which will update the working tree if pushing into the current branch. This option is intended for synchronizing working directories when one side is not easily accessible via interactive ssh (e.g. a live web site, hence the requirement that

the working directory be clean). This mode also comes in handy when developing inside a VM to test and fix code on different Operating Systems.

By default, "updateInstead" will refuse the push if the working tree or the index have any difference from the HEAD, but the `push-to-checkout` hook can be used to customize this. See githooks[5].

receive.denyNonFastForwards

If set to true, git-receive-pack will deny a ref update which is not a fast-forward. Use this to prevent such an update via a push, even if that push is forced. This configuration variable is set when initializing a shared repository.

receive.hideRefs

This variable is the same as `transfer.hideRefs`, but applies only to `receive-pack` (and so affects pushes, but not fetches). An attempt to update or delete a hidden ref by `git push` is rejected.

receive.updateServerInfo

If set to true, git-receive-pack will run git-update-server-info after receiving data from git-push and updating refs.

receive.shallowUpdate

If set to true, .git/shallow can be updated when new refs require new shallow roots. Otherwise those refs are rejected.

remote.pushDefault

The remote to push to by default. Overrides `branch.<name>.remote` for all branches, and is overridden by `branch.<name>.pushRemote` for specific branches.

remote.<name>.url

The URL of a remote repository. See git-fetch[1] or git-push[1].

remote.<name>.pushurl

The push URL of a remote repository. See git-push[1].

remote.<name>.proxy

For remotes that require curl (http, https and ftp), the URL to the proxy to use for that remote. Set to the empty string to disable proxying for that remote.

remote.<name>.proxyAuthMethod

For remotes that require curl (http, https and ftp), the method to use for authenticating against the proxy in use (probably set in `remote.<name>.proxy`). See `http.proxyAuthMethod`.

remote.<name>.fetch

The default set of "refspec" for git-fetch[1]. See git-fetch[1].

remote.<name>.push

The default set of "refspec" for git-push[1]. See git-push[1].

remote.<name>.mirror

If true, pushing to this remote will automatically behave as if the `--mirror` option was given on the command line.

remote.<name>.skipDefaultUpdate

If true, this remote will be skipped by default when updating using git-fetch[1] or the `update` subcommand of git-remote[1].

remote.<name>.skipFetchAll

If true, this remote will be skipped by default when updating using git-fetch[1] or the `update` subcommand of git-remote[1].

remote.<name>.receivepack

The default program to execute on the remote side when pushing. See option --receive-pack of git-push[1].

remote.<name>.uploadpack

The default program to execute on the remote side when fetching. See option --upload-pack of git-fetch-pack[1].

remote.<name>.tagOpt

Setting this value to --no-tags disables automatic tag following when fetching from remote <name>. Setting it to --tags will fetch every tag from remote <name>, even if they are not reachable from remote branch heads. Passing these flags directly to git-fetch[1] can override this setting. See options --tags and --no-tags of git-fetch[1].

remote.<name>.vcs

Setting this to a value <vcs> will cause Git to interact with the remote with the git-remote-<vcs> helper.

remote.<name>.prune

When set to true, fetching from this remote by default will also remove any remote-tracking references that no longer exist on the remote (as if the `--prune` option was given on the command line). Overrides `fetch.prune` settings, if any.

remotes.<group>

The list of remotes which are fetched by "git remote update <group>". See git-remote[1].

repack.useDeltaBaseOffset

By default, git-repack[1] creates packs that use delta-base offset. If you need to share your repository with Git older than version 1.4.4, either directly or via a dumb protocol such as http, then you need to set this option to "false" and repack. Access from old Git versions over the native protocol are unaffected by this option.

repack.packKeptObjects

If set to true, makes `git repack` act as if `--pack-kept-objects` was passed. See git-repack[1] for details. Defaults to `false` normally, but `true` if a bitmap index is being written (either via `--write-bitmap-index` or `repack.writeBitmaps`).

repack.writeBitmaps

When true, git will write a bitmap index when packing all objects to disk (e.g., when `git repack -a` is run). This index can speed up the "counting objects" phase of subsequent packs created for clones and fetches, at the cost of some disk space and extra time spent on the initial repack. Defaults to false.

rerere.autoUpdate

When set to true, `git-rerere` updates the index with the resulting contents after it cleanly resolves conflicts using previously recorded resolution. Defaults to false.

rerere.enabled

Activate recording of resolved conflicts, so that identical conflict hunks can be resolved automatically, should they be encountered again. By default, git-rerere[1] is enabled if there is an `rr-cache` directory under the `$GIT_DIR` , e.g. if "rerere" was previously used in the repository.

sendemail.identity

A configuration identity. When given, causes values in the *sendemail.<identity>* subsection to take precedence over values in the *sendemail* section. The default identity is the value of *sendemail.identity*.

sendemail.smtpEncryption

See git-send-email[1] for description. Note that this setting is not subject to the *identity* mechanism.

sendemail.smtpssl (deprecated)

Deprecated alias for *sendemail.smtpEncryption = ssl*.

sendemail.smtpsslcertpath

Path to ca-certificates (either a directory or a single file). Set it to an empty string to disable certificate verification.

sendemail.<identity>.*

Identity-specific versions of the *sendemail.** parameters found below, taking precedence over those when the this identity is selected, through command-line or *sendemail.identity*.

sendemail.aliasesFile

sendemail.aliasFileType

sendemail.annotate

sendemail.bcc

sendemail.cc

sendemail.ccCmd

sendemail.chainReplyTo

sendemail.confirm

sendemail.envelopeSender

sendemail.from

sendemail.multiEdit

sendemail.signedoffbycc

sendemail.smtpPass

sendemail.suppresscc

sendemail.suppressFrom

sendemail.to

sendemail.smtpDomain

sendemail.smtpServer

sendemail.smtpServerPort

sendemail.smtpServerOption

sendemail.smtpUser

sendemail.thread

sendemail.transferEncoding

sendemail.validate

sendemail.xmailer

See git-send-email[1] for description.

sendemail.signedoffcc (deprecated)

Deprecated alias for *sendemail.signedoffbycc*.

showbranch.default

The default set of branches for git-show-branch[1]. See git-show-branch[1].

status.relativePaths

By default, git-status[1] shows paths relative to the current directory. Setting this variable to `false` shows paths relative to the repository root (this was the default for Git prior to v1.5.4).

status.short

Set to true to enable --short by default in git-status[1]. The option --no-short takes precedence over this variable.

status.branch

Set to true to enable --branch by default in git-status[1]. The option --no-branch takes precedence over this variable.

status.displayCommentPrefix

If set to true, git-status[1] will insert a comment prefix before each output line (starting with `core.commentChar`, i.e. `#` by default). This was the behavior of git-status[1] in Git 1.8.4 and previous. Defaults to false.

status.showUntrackedFiles

By default, git-status[1] and git-commit[1] show files which are not currently tracked by Git. Directories which contain only untracked files, are shown with the directory name only. Showing untracked files means that Git needs to lstat() all the files in the whole repository, which might be slow on some systems. So, this variable controls how the commands displays the untracked files. Possible values are:

- `no` - Show no untracked files.
- `normal` - Show untracked files and directories.
- `all` - Show also individual files in untracked directories.

If this variable is not specified, it defaults to *normal*. This variable can be overridden with the -u|--untracked-files option of git-status[1] and git-commit[1].

status.submoduleSummary

Defaults to false. If this is set to a non zero number or true (identical to -1 or an unlimited number), the submodule summary will be enabled and a summary of commits for modified submodules will be shown (see --summary-limit option of git-submodule[1]). Please note that the summary output command will be suppressed for all submodules when `diff.ignoreSubmodules` is set to *all* or only for those submodules where `submodule.<name>.ignore=all`. The only exception to that rule is that status and commit will show staged submodule changes. To also view the summary for ignored submodules you can either use the --ignore-submodules=dirty command-line option or the *git submodule summary* command, which shows a similar output but does not honor these settings.

stash.showPatch

If this is set to true, the `git stash show` command without an option will show the stash in patch form. Defaults to false. See description of *show* command in git-stash[1].

stash.showStat

If this is set to true, the `git stash show` command without an option will show diffstat of the stash. Defaults to true. See description of *show* command in git-stash[1].

submodule.<name>.path

submodule.<name>.url

The path within this project and URL for a submodule. These variables are initially populated by *git submodule init*. See git-submodule[1] and gitmodules[5] for details.

submodule.<name>.update

The default update procedure for a submodule. This variable is populated by `git submodule init` from the gitmodules[5] file. See description of *update* command in git-submodule[1].

submodule.<name>.branch

The remote branch name for a submodule, used by `git submodule update --remote`. Set this option to override the value found in the `.gitmodules` file. See git-submodule[1] and gitmodules[5] for details.

submodule.<name>.fetchRecurseSubmodules

This option can be used to control recursive fetching of this submodule. It can be overridden by using the --[no-]recurse-submodules command-line option to "git fetch" and "git pull". This setting will override that from in the gitmodules[5] file.

submodule.<name>.ignore

Defines under what circumstances "git status" and the diff family show a submodule as modified. When set to "all", it will never be considered modified (but it will nonetheless show up in the output of status and commit when it has been staged), "dirty" will ignore all changes to the submodules work tree and takes only differences between the HEAD of the submodule and the commit recorded in the superproject into account. "untracked" will additionally let submodules with modified tracked files in their work tree show up. Using "none" (the default when this option is not set) also shows submodules that have untracked files in their work tree as changed. This setting overrides any setting made in .gitmodules for this submodule, both settings can be overridden on the command line by using the "--ignore-submodules" option. The *git submodule* commands are not affected by this setting.

tag.sort

This variable controls the sort ordering of tags when displayed by git-tag[1]. Without the "--sort=<value>" option provided, the value of this variable will be used as the default.

tar.umask

This variable can be used to restrict the permission bits of tar archive entries. The default is 0002, which turns off the world write bit. The special value "user" indicates that the archiving user’s umask will be used instead. See umask(2) and git-archive[1].

transfer.fsckObjects

When `fetch.fsckObjects` or `receive.fsckObjects` are not set, the value of this variable is used instead. Defaults to false.

transfer.hideRefs

String(s) `receive-pack` and `upload-pack` use to decide which refs to omit from their initial advertisements. Use more than one definition to specify multiple prefix strings. A ref that is under the hierarchies listed in the value of this variable is excluded, and is hidden when responding to `git push` or `git fetch`. See `receive.hideRefs` and `uploadpack.hideRefs` for program-specific versions of this config.

You may also include a `!` in front of the ref name to negate the entry, explicitly exposing it, even if an earlier entry marked it as hidden. If you have multiple hideRefs values, later entries override earlier ones (and entries in more-specific config files override less-specific ones).

If a namespace is in use, the namespace prefix is stripped from each reference before it is matched against `transfer.hiderefs` patterns. For example, if `refs/heads/master` is specified in `transfer.hideRefs` and the current namespace is `foo`, then `refs/namespaces/foo/refs/heads/master` is omitted from the advertisements but `refs/heads/master` and `refs/namespaces/bar/refs/heads/master` are still advertised as so-called "have" lines. In order to match refs before stripping, add a `^` in front of the ref name. If you combine `!` and `^`, `!` must be specified first.

transfer.unpackLimit

When `fetch.unpackLimit` or `receive.unpackLimit` are not set, the value of this variable is used instead. The default value is 100.

uploadarchive.allowUnreachable

If true, allow clients to use `git archive --remote` to request any tree, whether reachable from the ref tips or not. See the discussion in the `SECURITY` section of git-upload-archive[1] for more details. Defaults to `false`.

uploadpack.hideRefs

This variable is the same as `transfer.hideRefs`, but applies only to `upload-pack` (and so affects only fetches, not pushes). An attempt to fetch a hidden ref by `git fetch` will fail. See also `uploadpack.allowTipSHA1InWant`.

uploadpack.allowTipSHA1InWant

When `uploadpack.hideRefs` is in effect, allow `upload-pack` to accept a fetch request that asks for an object at the tip of a hidden ref (by default, such a request is rejected). see also `uploadpack.hideRefs`.

uploadpack.allowReachableSHA1InWant

Allow `upload-pack` to accept a fetch request that asks for an object that is reachable from any ref tip. However, note that calculating object reachability is computationally expensive. Defaults to `false`.

uploadpack.keepAlive

When `upload-pack` has started `pack-objects`, there may be a quiet period while `pack-objects` prepares the pack. Normally it would output progress information, but if `--quiet` was used for the fetch, `pack-objects` will output nothing at all until the pack data begins. Some clients and networks may consider the server to be hung and give up. Setting this option instructs `upload-pack` to send an empty keepalive packet every `uploadpack.keepAlive` seconds. Setting this option to 0 disables keepalive packets entirely. The default is 5 seconds.

url.<base>.insteadOf

Any URL that starts with this value will be rewritten to start, instead, with <base>. In cases where some site serves a large number of repositories, and serves them with multiple access methods, and some users need to use different access methods, this feature allows people to specify any of the equivalent URLs and have Git automatically rewrite the URL to the best alternative for the particular user, even for a never-before-seen repository on the site. When more than one insteadOf strings match a given URL, the longest match is used.

url.<base>.pushInsteadOf

Any URL that starts with this value will not be pushed to; instead, it will be rewritten to start with <base>, and the resulting URL will be pushed to. In cases where some site serves a large number of repositories, and serves them with multiple access methods, some of which do not allow push, this feature allows people to specify a pull-only URL and have Git automatically use an appropriate URL to push, even for a never-before-seen repository on the site. When more than one pushInsteadOf strings match a given URL, the longest match is used. If a remote has an explicit pushurl, Git will ignore this setting for that remote.

user.email

Your email address to be recorded in any newly created commits. Can be overridden by the *GIT_AUTHOR_EMAIL*, *GIT_COMMITTER_EMAIL*, and *EMAIL* environment variables. See git-commit-tree[1].

user.name

Your full name to be recorded in any newly created commits. Can be overridden by the *GIT_AUTHOR_NAME* and *GIT_COMMITTER_NAME* environment variables. See git-commit-tree[1].

user.useConfigOnly

Instruct Git to avoid trying to guess defaults for *user.email* and *user.name*, and instead retrieve the values only from the configuration. For example, if you have multiple email addresses and would like to use a different one for each repository, then with this configuration option set to `true` in the global config along with a name, Git will prompt you to set up an email before making new commits in a newly cloned repository. Defaults to `false`.

user.signingKey

If git-tag[1] or git-commit[1] is not selecting the key you want it to automatically when creating a signed tag or commit, you can override the default selection with this variable. This option is passed unchanged to gpg's --local-user parameter, so you may specify a key using any method that gpg supports.

versionsort.prereleaseSuffix

When version sort is used in git-tag[1], prerelease tags (e.g. "1.0-rc1") may appear after the main release "1.0". By specifying the suffix "-rc" in this variable, "1.0-rc1" will appear before "1.0".

This variable can be specified multiple times, once per suffix. The order of suffixes in the config file determines the sorting order (e.g. if "-pre" appears before "-rc" in the config file then 1.0-preXX is sorted before 1.0-rcXX). The sorting order between different suffixes is undefined if they are in multiple config files.

web.browser

Specify a web browser that may be used by some commands. Currently only git-instaweb[1] and git-help[1] may use it.

GIT

Part of the git[1] suite

help

NAME

git-help - Display help information about Git

SYNOPSIS

```
git help [-a|--all] [-g|--guide]
        [-i|--info|-m|--man|-w|--web] [COMMAND|GUIDE]
```

DESCRIPTION

With no options and no COMMAND or GUIDE given, the synopsis of the *git* command and a list of the most commonly used Git commands are printed on the standard output.

If the option *--all* or *-a* is given, all available commands are printed on the standard output.

If the option *--guide* or *-g* is given, a list of the useful Git guides is also printed on the standard output.

If a command, or a guide, is given, a manual page for that command or guide is brought up. The *man* program is used by default for this purpose, but this can be overridden by other options or configuration variables.

Note that `git --help ...` is identical to `git help ...` because the former is internally converted into the latter.

To display the git[1] man page, use `git help git`.

This page can be displayed with *git help help* or `git help --help`

OPTIONS

-a

--all

Prints all the available commands on the standard output. This option overrides any given command or guide name.

-g

--guides

Prints a list of useful guides on the standard output. This option overrides any given command or guide name.

-i

--info

Display manual page for the command in the *info* format. The *info* program will be used for that purpose.

-m

--man

Display manual page for the command in the *man* format. This option may be used to override a value set in the *help.format* configuration variable.

By default the *man* program will be used to display the manual page, but the *man.viewer* configuration variable may be used to choose other display programs (see below).

-w

--web

Display manual page for the command in the *web* (HTML) format. A web browser will be used for that purpose.

The web browser can be specified using the configuration variable *help.browser*, or *web.browser* if the former is not set. If none of these config variables is set, the *git web{litdd}browse* helper script (called by *git help*) will pick a suitable default. See git-web{litdd}browse[1] for more information about this.

CONFIGURATION VARIABLES

help.format

If no command-line option is passed, the *help.format* configuration variable will be checked. The following values are supported for this variable; they make *git help* behave as their corresponding command- line option:

- "man" corresponds to *-m*|*--man*,
- "info" corresponds to *-i*|*--info*,

- "web" or "html" correspond to *-w|--web*.

help.browser, web.browser and browser.<tool>.path

The *help.browser*, *web.browser* and *browser.<tool>.path* will also be checked if the *web* format is chosen (either by command-line option or configuration variable). See *-w|--web* in the OPTIONS section above and git-web{litdd}browse[1].

man.viewer

The *man.viewer* configuration variable will be checked if the *man* format is chosen. The following values are currently supported:

- "man": use the *man* program as usual,
- "woman": use *emacsclient* to launch the "woman" mode in emacs (this only works starting with emacsclient versions 22),
- "konqueror": use *kfmclient* to open the man page in a new konqueror tab (see *Note about konqueror* below).

Values for other tools can be used if there is a corresponding *man.<tool>.cmd* configuration entry (see below).

Multiple values may be given to the *man.viewer* configuration variable. Their corresponding programs will be tried in the order listed in the configuration file.

For example, this configuration:

```
[man]
        viewer = konqueror
        viewer = woman
```

will try to use konqueror first. But this may fail (for example, if DISPLAY is not set) and in that case emacs' woman mode will be tried.

If everything fails, or if no viewer is configured, the viewer specified in the GIT*MAN_VIEWER environment variable will be tried. If that fails too, the _man* program will be tried anyway.

man.<tool>.path

You can explicitly provide a full path to your preferred man viewer by setting the configuration variable *man.<tool>.path*. For example, you can configure the absolute path to konqueror by setting *man.konqueror.path*. Otherwise, *git help* assumes the tool is available in PATH.

man.<tool>.cmd

When the man viewer, specified by the *man.viewer* configuration variables, is not among the supported ones, then the corresponding *man.<tool>.cmd* configuration variable will be looked up. If this variable exists then the specified tool will be treated as a custom command and a shell eval will be used to run the command with the man page passed as arguments.

Note about konqueror

When *konqueror* is specified in the *man.viewer* configuration variable, we launch *kfmclient* to try to open the man page on an already opened konqueror in a new tab if possible.

For consistency, we also try such a trick if *man.konqueror.path* is set to something like *A_PATH_TO/konqueror*. That means we will try to launch *A_PATH_TO/kfmclient* instead.

If you really want to use *konqueror*, then you can use something like the following:

```
        [man]
                viewer = konq

        [man "konq"]
                cmd = A_PATH_TO/konqueror
```

Note about git config --global

Note that all these configuration variables should probably be set using the *--global* flag, for example like this:

```
$ git config --global help.format web
$ git config --global web.browser firefox
```

as they are probably more user specific than repository specific. See git-config[1] for more information about this.

GIT

Part of the git[1] suite

Getting and Creating Projects

init

NAME

git-init - Create an empty Git repository or reinitialize an existing one

SYNOPSIS

```
git init [-q | --quiet] [--bare] [--template=<template_directory>]
        [--separate-git-dir <git dir>]
        [--shared[=<permissions>]] [directory]
```

DESCRIPTION

This command creates an empty Git repository - basically a `.git` directory with subdirectories for `objects`, `refs/heads`, `refs/tags`, and template files. An initial `HEAD` file that references the HEAD of the master branch is also created.

If the `$GIT_DIR` environment variable is set then it specifies a path to use instead of `./.git` for the base of the repository.

If the object storage directory is specified via the `$GIT_OBJECT_DIRECTORY` environment variable then the sha1 directories are created underneath - otherwise the default `$GIT_DIR/objects` directory is used.

Running *git init* in an existing repository is safe. It will not overwrite things that are already there. The primary reason for rerunning *git init* is to pick up newly added templates (or to move the repository to another place if --separate-git-dir is given).

OPTIONS

-q

--quiet

Only print error and warning messages; all other output will be suppressed.

--bare

Create a bare repository. If GIT_DIR environment is not set, it is set to the current working directory.

--template=<template_directory>

Specify the directory from which templates will be used. (See the "TEMPLATE DIRECTORY" section below.)

--separate-git-dir=<git dir>

Instead of initializing the repository as a directory to either `$GIT_DIR` or `./.git/`, create a text file there containing the path to the actual repository. This file acts as filesystem-agnostic Git symbolic link to the repository.

If this is reinitialization, the repository will be moved to the specified path.

--shared[=(false|true|umask|group|all|world|everybody|0xxx)]

Specify that the Git repository is to be shared amongst several users. This allows users belonging to the same group to push into that repository. When specified, the config variable "core.sharedRepository" is set so that files and directories under `$GIT_DIR` are created with the requested permissions. When not specified, Git will use permissions reported by umask(2).

The option can have the following values, defaulting to *group* if no value is given:

umask (or *false*)

Use permissions reported by umask(2). The default, when `--shared` is not specified.

group (or *true*)

Make the repository group-writable, (and g+sx, since the git group may be not the primary group of all users). This is used to loosen the permissions of an otherwise safe umask(2) value. Note that the umask still applies to the other permission bits (e.g. if umask is *0022*, using *group* will not remove read privileges from other (non-group) users). See *0xxx* for how to exactly specify the repository permissions.

all (or *world* or *everybody*)

Same as *group*, but make the repository readable by all users.

0xxx

0xxx is an octal number and each file will have mode *0xxx*. *0xxx* will override users' umask(2) value (and not only loosen permissions as *group* and *all* does). *0640* will create a repository which is group-readable, but not group-writable or accessible to others. *0660* will

create a repo that is readable and writable to the current user and group, but inaccessible to others.

By default, the configuration flag `receive.denyNonFastForwards` is enabled in shared repositories, so that you cannot force a non fast-forwarding push into it.

If you provide a *directory*, the command is run inside it. If this directory does not exist, it will be created.

TEMPLATE DIRECTORY

The template directory contains files and directories that will be copied to the `$GIT_DIR` after it is created.

The template directory will be one of the following (in order):

- the argument given with the `--template` option;
- the contents of the `$GIT_TEMPLATE_DIR` environment variable;
- the `init.templateDir` configuration variable; or
- the default template directory: `/usr/share/git-core/templates`.

The default template directory includes some directory structure, suggested "exclude patterns" (see gitignore[5]), and sample hook files (see githooks[5]).

EXAMPLES

Start a new Git repository for an existing code base

```
$ cd /path/to/my/codebase
$ git init      (1)
$ git add .     (2)
$ git commit    (3)
```

1. Create a /path/to/my/codebase/.git directory.
2. Add all existing files to the index.
3. Record the pristine state as the first commit in the history.

GIT

Part of the git[1] suite

clone

NAME

git-clone - Clone a repository into a new directory

SYNOPSIS

```
git clone [--template=<template_directory>]
      [-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror]
      [-o <name>] [-b <name>] [-u <upload-pack>] [--reference <repository>]
      [--dissociate] [--separate-git-dir <git dir>]
      [--depth <depth>] [--[no-]single-branch]
      [--recursive | --recurse-submodules] [--] <repository>
      [<directory>]
```

DESCRIPTION

Clones a repository into a newly created directory, creates remote-tracking branches for each branch in the cloned repository (visible using `git branch -r`), and creates and checks out an initial branch that is forked from the cloned repository's currently active branch.

After the clone, a plain `git fetch` without arguments will update all the remote-tracking branches, and a `git pull` without arguments will in addition merge the remote master branch into the current master branch, if any (this is untrue when "--single-branch" is given; see below).

This default configuration is achieved by creating references to the remote branch heads under `refs/remotes/origin` and by initializing `remote.origin.url` and `remote.origin.fetch` configuration variables.

OPTIONS

--local

-l

When the repository to clone from is on a local machine, this flag bypasses the normal "Git aware" transport mechanism and clones the repository by making a copy of HEAD and everything under objects and refs directories. The files under `.git/objects/` directory are hardlinked to save space when possible.

If the repository is specified as a local path (e.g., `/path/to/repo`), this is the default, and --local is essentially a no-op. If the repository is specified as a URL, then this flag is ignored (and we never use the local optimizations). Specifying `--no-local` will override the default when `/path/to/repo` is given, using the regular Git transport instead.

--no-hardlinks

Force the cloning process from a repository on a local filesystem to copy the files under the `.git/objects` directory instead of using hardlinks. This may be desirable if you are trying to make a back-up of your repository.

--shared

-s

When the repository to clone is on the local machine, instead of using hard links, automatically setup `.git/objects/info/alternates` to share the objects with the source repository. The resulting repository starts out without any object of its own.

NOTE: this is a possibly dangerous operation; do **not** use it unless you understand what it does. If you clone your repository using this option and then delete branches (or use any other Git command that makes any existing commit unreferenced) in the source repository, some objects may become unreferenced (or dangling). These objects may be removed by normal Git operations (such as `git commit`) which automatically call `git gc --auto`. (See git-gc[1].) If these objects are removed and were referenced by the cloned repository, then the cloned repository will become corrupt.

Note that running `git repack` without the `-l` option in a repository cloned with `-s` will copy objects from the source repository into a pack in the cloned repository, removing the disk space savings of `clone -s`. It is safe, however, to run `git gc`, which uses the `-l` option by default.

If you want to break the dependency of a repository cloned with `-s` on its source repository, you can simply run `git repack -a` to copy all objects from the source repository into a pack in the cloned repository.

--reference <repository>

If the reference repository is on the local machine, automatically setup `.git/objects/info/alternates` to obtain objects from the reference repository. Using an already existing repository as an alternate will require fewer objects to be copied from the repository being cloned, reducing network and local storage costs.

NOTE: see the NOTE for the `--shared` option, and also the `--dissociate` option.

--dissociate

Borrow the objects from reference repositories specified with the `--reference` options only to reduce network transfer, and stop borrowing from them after a clone is made by making necessary local copies of borrowed objects. This option can also be used when cloning locally from a repository that already borrows objects from another repository—the new repository will borrow objects from the same repository, and this option can be used to stop the borrowing.

--quiet

-q

Operate quietly. Progress is not reported to the standard error stream.

--verbose

-v

Run verbosely. Does not affect the reporting of progress status to the standard error stream.

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

--no-checkout

-n

No checkout of HEAD is performed after the clone is complete.

--bare

Make a *bare* Git repository. That is, instead of creating `<directory>` and placing the administrative files in `<directory>/.git`, make the `<directory>` itself the `$GIT_DIR`. This obviously implies the `-n` because there is nowhere to check out the working tree. Also the branch heads at the remote are copied directly to corresponding local branch heads, without mapping them to `refs/remotes/origin/`. When this option is used, neither remote-tracking branches nor the related configuration variables are created.

--mirror

Set up a mirror of the source repository. This implies `--bare`. Compared to `--bare`, `--mirror` not only maps local branches of the source to local branches of the target, it maps all refs (including remote-tracking branches, notes etc.) and sets up a refspec configuration such that all these refs are overwritten by a `git remote update` in the target repository.

--origin <name>

-o <name>

Instead of using the remote name `origin` to keep track of the upstream repository, use `<name>`.

--branch <name>

-b <name>

Instead of pointing the newly created HEAD to the branch pointed to by the cloned repository’s HEAD, point to `<name>` branch instead. In a non-bare repository, this is the branch that will be checked out. `--branch` can also take tags and detaches the HEAD at that commit in the resulting repository.

--upload-pack <upload-pack>

-u <upload-pack>

When given, and the repository to clone from is accessed via ssh, this specifies a non-default path for the command run on the other end.

--template=<template_directory>

Specify the directory from which templates will be used; (See the "TEMPLATE DIRECTORY" section of git-init[1].)

--config <key>=<value>

-c <key>=<value>

Set a configuration variable in the newly-created repository; this takes effect immediately after the repository is initialized, but before the remote history is fetched or any files checked out. The key is in the same format as expected by git-config[1] (e.g., `core.eol=true`). If multiple values are given for the same key, each value will be written to the config file. This makes it safe, for example, to add additional fetch refspecs to the origin remote.

--depth <depth>

Create a *shallow* clone with a history truncated to the specified number of commits. Implies `--single-branch` unless `--no-single-branch` is given to fetch the histories near the tips of all branches.

--[no-]single-branch

Clone only the history leading to the tip of a single branch, either specified by the `--branch` option or the primary branch remote’s `HEAD` points at. Further fetches into the resulting repository will only update the remote-tracking branch for the branch this option was used for the initial cloning. If the HEAD at the remote did not point at any branch when `--single-branch` clone was made, no remote-tracking branch is created.

--recursive

--recurse-submodules

After the clone is created, initialize all submodules within, using their default settings. This is equivalent to running `git submodule update --init --recursive` immediately after the clone is finished. This option is ignored if the cloned repository does not have a worktree/checkout (i.e. if any of `--no-checkout` / `-n` , `--bare` , or `--mirror` is given)

--separate-git-dir=<git dir>

Instead of placing the cloned repository where it is supposed to be, place the cloned repository at the specified directory, then make a filesystem-agnostic Git symbolic link to there. The result is Git repository can be separated from working tree.

<repository>

The (possibly remote) repository to clone from. See the URLS section below for more information on specifying repositories.

<directory>

The name of a new directory to clone into. The "humanish" part of the source repository is used if no directory is explicitly given (`repo` for `/path/to/repo.git` and `foo` for `host.xz:foo/.git`). Cloning into an existing directory is only allowed if the directory is empty.

GIT URLS

In general, URLs contain information about the transport protocol, the address of the remote server, and the path to the repository. Depending on the transport protocol, some of this information may be absent.

Git supports ssh, git, http, and https protocols (in addition, ftp, and ftps can be used for fetching, but this is inefficient and deprecated; do not use it).

The native transport (i.e. git:// URL) does no authentication and should be used with caution on unsecured networks.

The following syntaxes may be used with them:

- ssh://[user@]host.xz[:port]/path/to/repo.git/
- git://host.xz[:port]/path/to/repo.git/
- http[s]://host.xz[:port]/path/to/repo.git/
- ftp[s]://host.xz[:port]/path/to/repo.git/

An alternative scp-like syntax may also be used with the ssh protocol:

- [user@]host.xz:path/to/repo.git/

This syntax is only recognized if there are no slashes before the first colon. This helps differentiate a local path that contains a colon. For example the local path `foo:bar` could be specified as an absolute path or `./foo:bar` to avoid being misinterpreted as an ssh url.

The ssh and git protocols additionally support ~username expansion:

- ssh://[user@]host.xz[:port]/~[user]/path/to/repo.git/
- git://host.xz[:port]/~[user]/path/to/repo.git/
- [user@]host.xz:/~[user]/path/to/repo.git/

For local repositories, also supported by Git natively, the following syntaxes may be used:

- /path/to/repo.git/
- file:///path/to/repo.git/

These two syntaxes are mostly equivalent, except the former implies --local option.

When Git doesn’t know how to handle a certain transport protocol, it attempts to use the *remote-<transport>* remote helper, if one exists. To explicitly request a remote helper, the following syntax may be used:

- <transport>::<address>

where <address> may be a path, a server and path, or an arbitrary URL-like string recognized by the specific remote helper being invoked. See gitremote-helpers[1] for details.

If there are a large number of similarly-named remote repositories and you want to use a different format for them (such that the URLs you use will be rewritten into URLs that work), you can create a configuration section of the form:

```
[url "<actual url base>"]
    insteadOf = <other url base>
```

For example, with this:

```
[url "git://git.host.xz/"]
    insteadOf = host.xz:/path/to/
    insteadOf = work:
```

a URL like "work:repo.git" or like "host.xz:/path/to/repo.git" will be rewritten in any context that takes a URL to be "git://git.host.xz/repo.git".

If you want to rewrite URLs for push only, you can create a configuration section of the form:

```
[url "<actual url base>"]
    pushInsteadOf = <other url base>
```

For example, with this:

```
[url "ssh://example.org/"]
    pushInsteadOf = git://example.org/
```

a URL like "git://example.org/path/to/repo.git" will be rewritten to "ssh://example.org/path/to/repo.git" for pushes, but pulls will still use the original URL.

Examples

- Clone from upstream:

```
$ git clone git://git.kernel.org/pub/scm/.../linux.git my-linux
$ cd my-linux
$ make
```

- Make a local clone that borrows from the current directory, without checking things out:

```
$ git clone -l -s -n . ../copy
$ cd ../copy
$ git show-branch
```

- Clone from upstream while borrowing from an existing local directory:

```
$ git clone --reference /git/linux.git \
    git://git.kernel.org/pub/scm/.../linux.git \
    my-linux
$ cd my-linux
```

- Create a bare repository to publish your changes to the public:

```
$ git clone --bare -l /home/proj/.git /pub/scm/proj.git
```

GIT

Part of the git[1] suite

Basic Snapshotting

add

NAME

git-add - Add file contents to the index

SYNOPSIS

```
git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch |
	  [--edit | -e] [--[no-]all | --[no-]ignore-removal | [--update | -u]]
	  [--intent-to-add | -N] [--refresh] [--ignore-errors] [--ignore-missing]
	  [--] [<pathspec>…]
```

DESCRIPTION

This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit. It typically adds the current content of existing paths as a whole, but with some options it can also be used to add content with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.

The "index" holds a snapshot of the content of the working tree, and it is this snapshot that is taken as the contents of the next commit. Thus after making any changes to the working tree, and before running the commit command, you must use the `add` command to add any new or modified files to the index.

This command can be performed multiple times before a commit. It only adds the content of the specified file(s) at the time the add command is run; if you want subsequent changes included in the next commit, then you must run `git add` again to add the new content to the index.

The `git status` command can be used to obtain a summary of which files have changes that are staged for the next commit.

The `git add` command will not add ignored files by default. If any ignored files were explicitly specified on the command line, `git add` will fail with a list of ignored files. Ignored files reached by directory recursion or filename globbing performed by Git (quote your globs before the shell) will be silently ignored. The *git add* command can be used to add ignored files with the `-f` (force) option.

Please see git-commit[1] for alternative ways to add content to a commit.

OPTIONS

<pathspec>…

Files to add content from. Fileglobs (e.g. `*.c`) can be given to add all matching files. Also a leading directory name (e.g. `dir` to add `dir/file1` and `dir/file2`) can be given to update the index to match the current state of the directory as a whole (e.g. specifying `dir` will record not just a file `dir/file1` modified in the working tree, a file `dir/file2` added to the working tree, but also a file `dir/file3` removed from the working tree. Note that older versions of Git used to ignore removed files; use `--no-all` option if you want to add modified or new files but ignore removed ones.

-n

--dry-run

Don’t actually add the file(s), just show if they exist and/or will be ignored.

-v

--verbose

Be verbose.

-f

--force

Allow adding otherwise ignored files.

-i

--interactive

Add modified contents in the working tree interactively to the index. Optional path arguments may be supplied to limit operation to a subset of the working tree. See “Interactive mode” for details.

-p

--patch

Interactively choose hunks of patch between the index and the work tree and add them to the index. This gives the user a chance to review the difference before adding modified contents to the index.

This effectively runs `add --interactive`, but bypasses the initial command menu and directly jumps to the `patch` subcommand. See “Interactive mode” for details.

-e

--edit

Open the diff vs. the index in an editor and let the user edit it. After the editor was closed, adjust the hunk headers and apply the patch to the index.

The intent of this option is to pick and choose lines of the patch to apply, or even to modify the contents of lines to be staged. This can be quicker and more flexible than using the interactive hunk selector. However, it is easy to confuse oneself and create a patch that does not apply to the index. See EDITING PATCHES below.

-u

--update

Update the index just where it already has an entry matching <pathspec>. This removes as well as modifies index entries to match the working tree, but adds no new files.

If no <pathspec> is given when `-u` option is used, all tracked files in the entire working tree are updated (old versions of Git used to limit the update to the current directory and its subdirectories).

-A

--all

--no-ignore-removal

Update the index not only where the working tree has a file matching <pathspec> but also where the index already has an entry. This adds, modifies, and removes index entries to match the working tree.

If no <pathspec> is given when `-A` option is used, all files in the entire working tree are updated (old versions of Git used to limit the update to the current directory and its subdirectories).

--no-all

--ignore-removal

Update the index by adding new files that are unknown to the index and files modified in the working tree, but ignore files that have been removed from the working tree. This option is a no-op when no <pathspec> is used.

This option is primarily to help users who are used to older versions of Git, whose "git add <pathspec>…" was a synonym for "git add --no-all <pathspec>…", i.e. ignored removed files.

-N

--intent-to-add

Record only the fact that the path will be added later. An entry for the path is placed in the index with no content. This is useful for, among other things, showing the unstaged content of such files with `git diff` and committing them with `git commit -a`.

--refresh

Don’t add the file(s), but only refresh their stat() information in the index.

--ignore-errors

If some files could not be added because of errors indexing them, do not abort the operation, but continue adding the others. The command shall still exit with non-zero status. The configuration variable `add.ignoreErrors` can be set to true to make this the default behaviour.

--ignore-missing

This option can only be used together with --dry-run. By using this option the user can check if any of the given files would be ignored, no matter if they are already present in the work tree or not.

--

This option can be used to separate command-line options from the list of files, (useful when filenames might be mistaken for command-line options).

Configuration

The optional configuration variable `core.excludesFile` indicates a path to a file containing patterns of file names to exclude from git-add, similar to $GIT_DIR/info/exclude. Patterns in the exclude file are used in addition to those in info/exclude. See gitignore[5].

EXAMPLES

- Adds content from all `*.txt` files under `Documentation` directory and its subdirectories:

```
$ git add Documentation/\*.txt
```

Note that the asterisk `*` is quoted from the shell in this example; this lets the command include the files from subdirectories of `Documentation/` directory.

- Considers adding content from all git-*.sh scripts:

```
$ git add git-*.sh
```

Because this example lets the shell expand the asterisk (i.e. you are listing the files explicitly), it does not consider `subdir/git-foo.sh`.

Interactive mode

When the command enters the interactive mode, it shows the output of the *status* subcommand, and then goes into its interactive command loop.

The command loop shows the list of subcommands available, and gives a prompt "What now> ". In general, when the prompt ends with a single >, you can pick only one of the choices given and type return, like this:

```
    *** Commands ***
      1: status       2: update       3: revert       4: add untracked
      5: patch        6: diff         7: quit         8: help
    What now> 1
```

You also could say `s` or `sta` or `status` above as long as the choice is unique.

The main command loop has 6 subcommands (plus help and quit).

status

This shows the change between HEAD and index (i.e. what will be committed if you say `git commit`), and between index and working tree files (i.e. what you could stage further before `git commit` using `git add`) for each path. A sample output looks like this:

```
              staged     unstaged path
     1:       binary      nothing foo.png
     2:     +403/-35        +1/-1 git-add--interactive.perl
```

It shows that foo.png has differences from HEAD (but that is binary so line count cannot be shown) and there is no difference between indexed copy and the working tree version (if the working tree version were also different, *binary* would have been shown in place of *nothing*).

The other file, git-add{litdd}interactive.perl, has 403 lines added and 35 lines deleted if you commit what is in the index, but working tree file has further modifications (one addition and one deletion).

update

This shows the status information and issues an "Update>>" prompt. When the prompt ends with double >>, you can make more than one selection, concatenated with whitespace or comma. Also you can say ranges. E.g. "2-5 7,9" to choose 2,3,4,5,7,9 from the list. If the second number in a range is omitted, all remaining patches are taken. E.g. "7-" to choose 7,8,9 from the list. You can say * to choose everything.

What you chose are then highlighted with *, like this:

```
           staged     unstaged path
  1:       binary      nothing foo.png
* 2:     +403/-35        +1/-1 git-add--interactive.perl
```

To remove selection, prefix the input with `-` like this:

```
Update&gt;&gt; -2
```

After making the selection, answer with an empty line to stage the contents of working tree files for selected paths in the index.

revert

This has a very similar UI to *update*, and the staged information for selected paths are reverted to that of the HEAD version. Reverting new paths makes them untracked.

add untracked

This has a very similar UI to *update* and *revert*, and lets you add untracked paths to the index.

patch

This lets you choose one path out of a *status* like selection. After choosing the path, it presents the diff between the index and the working tree file and asks you if you want to stage the change of each hunk. You can select one of the following options and type return:

```
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

After deciding the fate for all hunks, if there is any hunk that was chosen, the index is updated with the selected hunks.

You can omit having to type return here, by setting the configuration variable `interactive.singleKey` to `true`.

diff

This lets you review what will be committed (i.e. between HEAD and index).

EDITING PATCHES

Invoking `git add -e` or selecting `e` from the interactive hunk selector will open a patch in your editor; after the editor exits, the result is applied to the index. You are free to make arbitrary changes to the patch, but note that some changes may have confusing results, or even result in a patch that cannot be applied. If you want to abort the operation entirely (i.e., stage nothing new in the index), simply delete all lines of the patch. The list below describes some common things you may see in a patch, and which editing operations make sense on them.

added content

Added content is represented by lines beginning with "+". You can prevent staging any addition lines by deleting them.

removed content

Removed content is represented by lines beginning with "-". You can prevent staging their removal by converting the "-" to a " " (space).

modified content

Modified content is represented by "-" lines (removing the old content) followed by "+" lines (adding the replacement content). You can prevent staging the modification by converting "-" lines to " ", and removing "+" lines. Beware that modifying only half of the pair is likely to introduce confusing changes to the index.

There are also more complex operations that can be performed. But beware that because the patch is applied only to the index and not the working tree, the working tree will appear to "undo" the change in the index. For example, introducing a new line into the index that is in neither the HEAD nor the working tree will stage the new line for commit, but the line will appear to be reverted in the working tree.

Avoid using these constructs, or do so with extreme caution.

removing untouched content

Content which does not differ between the index and working tree may be shown on context lines, beginning with a " " (space). You can stage context lines for removal by converting the space to a "-". The resulting working tree file will appear to re-add the content.

modifying existing content

One can also modify context lines by staging them for removal (by converting " " to "-") and adding a "+" line with the new content. Similarly, one can modify "+" lines for existing additions or modifications. In all cases, the new modification will appear reverted in the working tree.

new content

You may also add new content that does not exist in the patch; simply add new lines, each starting with "+". The addition will appear reverted in the working tree.

There are also several operations which should be avoided entirely, as they will make the patch impossible to apply:

- adding context (" ") or removal ("-") lines
- deleting context or removal lines
- modifying the contents of context or removal lines

SEE ALSO

git-status[1] git-rm[1] git-reset[1] git-mv[1] git-commit[1] git-update-index[1]

GIT

Part of the git[1] suite

status

NAME

git-status - Show the working tree status

SYNOPSIS

```
git status [<options>…] [--] [<pathspec>…]
```

DESCRIPTION

Displays paths that have differences between the index file and the current HEAD commit, paths that have differences between the working tree and the index file, and paths in the working tree that are not tracked by Git (and are not ignored by gitignore[5]). The first are what you *would* commit by running `git commit`; the second and third are what you *could* commit by running *git add* before running `git commit`.

OPTIONS

-s

--short

Give the output in the short-format.

-b

--branch

Show the branch and tracking info even in short-format.

--porcelain

Give the output in an easy-to-parse format for scripts. This is similar to the short output, but will remain stable across Git versions and regardless of user configuration. See below for details.

--long

Give the output in the long-format. This is the default.

-v

--verbose

In addition to the names of files that have been changed, also show the textual changes that are staged to be committed (i.e., like the output of `git diff --cached`). If `-v` is specified twice, then also show the changes in the working tree that have not yet been staged (i.e., like the output of `git diff`).

-u[<mode>]

--untracked-files[=<mode>]

Show untracked files.

The mode parameter is used to specify the handling of untracked files. It is optional: it defaults to *all*, and if specified, it must be stuck to the option (e.g. `-uno`, but not `-u no`).

The possible options are:

- *no* - Show no untracked files.
- *normal* - Shows untracked files and directories.
- *all* - Also shows individual files in untracked directories.

When `-u` option is not used, untracked files and directories are shown (i.e. the same as specifying `normal`), to help you avoid forgetting to add newly created files. Because it takes extra work to find untracked files in the filesystem, this mode may take some time in a large working tree. Consider enabling untracked cache and split index if supported (see `git update-index --untracked-cache` and `git update-index --split-index`), Otherwise you can use `no` to have `git status` return more quickly without showing untracked files.

The default can be changed using the status.showUntrackedFiles configuration variable documented in git-config[1].

--ignore-submodules[=<when>]

Ignore changes to submodules when looking for changes. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in git-config[1] or gitmodules[5]. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only

changes to the commits stored in the superproject are shown (this was the behavior before 1.7.0). Using "all" hides all changes to submodules (and suppresses the output of submodule summaries when the config option `status.submoduleSummary` is set).

--ignored

Show ignored files as well.

-z

Terminate entries with NUL, instead of LF. This implies the `--porcelain` output format if no other format is given.

--column[=<options>]

--no-column

Display untracked files in columns. See configuration variable column.status for option syntax.`--column` and `--no-column` without options are equivalent to *always* and *never* respectively.

OUTPUT

The output from this command is designed to be used as a commit template comment. The default, long format, is designed to be human readable, verbose and descriptive. Its contents and format are subject to change at any time.

The paths mentioned in the output, unlike many other Git commands, are made relative to the current directory if you are working in a subdirectory (this is on purpose, to help cutting and pasting). See the status.relativePaths config option below.

Short Format

In the short-format, the status of each path is shown as

```
XY PATH1 -> PATH2
```

where `PATH1` is the path in the `HEAD`, and the " `-> PATH2` " part is shown only when `PATH1` corresponds to a different path in the index/worktree (i.e. the file is renamed). The `XY` is a two-letter status code.

The fields (including the `->`) are separated from each other by a single space. If a filename contains whitespace or other nonprintable characters, that field will be quoted in the manner of a C string literal: surrounded by ASCII double quote (34) characters, and with

interior special characters backslash-escaped.

For paths with merge conflicts, `X` and `Y` show the modification states of each side of the merge. For paths that do not have merge conflicts, `X` shows the status of the index, and `Y` shows the status of the work tree. For untracked paths, `XY` are `??`. Other status codes can be interpreted as follows:

- ' ' = unmodified
- *M* = modified
- *A* = added
- *D* = deleted
- *R* = renamed
- *C* = copied
- *U* = updated but unmerged

Ignored files are not listed, unless `--ignored` option is in effect, in which case `XY` are `!!`.

```
X          Y     Meaning
-------------------------------------------------
          [MD]   not updated
M        [ MD]   updated in index
A        [ MD]   added to index
D         [ M]   deleted from index
R        [ MD]   renamed in index
C        [ MD]   copied in index
[MARC]           index and work tree matches
[ MARC]     M    work tree changed since index
[ MARC]     D    deleted in work tree
-------------------------------------------------
D           D    unmerged, both deleted
A           U    unmerged, added by us
U           D    unmerged, deleted by them
U           A    unmerged, added by them
D           U    unmerged, deleted by us
A           A    unmerged, both added
U           U    unmerged, both modified
-------------------------------------------------
?           ?    untracked
!           !    ignored
-------------------------------------------------
```

If -b is used the short-format status is preceded by a line

branchname tracking info

Porcelain Format

The porcelain format is similar to the short format, but is guaranteed not to change in a backwards-incompatible way between Git versions or based on user configuration. This makes it ideal for parsing by scripts. The description of the short format above also describes the porcelain format, with a few exceptions:

1. The user's color.status configuration is not respected; color will always be off.
2. The user's status.relativePaths configuration is not respected; paths shown will always be relative to the repository root.

There is also an alternate -z format recommended for machine parsing. In that format, the status field is the same, but some other things change. First, the -> is omitted from rename entries and the field order is reversed (e.g *from -> to* becomes *to from*). Second, a NUL (ASCII 0) follows each filename, replacing space as a field separator and the terminating newline (but a space still separates the status field from the first filename). Third, filenames containing special characters are not specially formatted; no quoting or backslash-escaping is performed.

CONFIGURATION

The command honors `color.status` (or `status.color` — they mean the same thing and the latter is kept for backward compatibility) and `color.status.<slot>` configuration variables to colorize its output.

If the config variable `status.relativePaths` is set to false, then all paths shown are relative to the repository root, not to the current directory.

If `status.submoduleSummary` is set to a non zero number or true (identical to -1 or an unlimited number), the submodule summary will be enabled for the long format and a summary of commits for modified submodules will be shown (see --summary-limit option of git-submodule[1]). Please note that the summary output from the status command will be suppressed for all submodules when `diff.ignoreSubmodules` is set to *all* or only for those submodules where `submodule.<name>.ignore=all`. To also view the summary for ignored submodules you can either use the --ignore-submodules=dirty command line option or the *git submodule summary* command, which shows a similar output but does not honor these settings.

SEE ALSO

gitignore[5]

GIT

Part of the git[1] suite

diff

NAME

git-diff - Show changes between commits, commit and working tree, etc

SYNOPSIS

```
git diff [options] [<commit>] [--] [<path>…]
git diff [options] --cached [<commit>] [--] [<path>…]
git diff [options] <commit> <commit> [--] [<path>…]
git diff [options] <blob> <blob>
git diff [options] [--no-index] [--] <path> <path>
```

DESCRIPTION

Show changes between the working tree and the index or a tree, changes between the index and a tree, changes between two trees, changes between two blob objects, or changes between two files on disk.

git diff [--options] [--] [<path>…]

This form is to view the changes you made relative to the index (staging area for the next commit). In other words, the differences are what you *could* tell Git to further add to the index but you still haven't. You can stage these changes by using git-add[1].

git diff --no-index [--options] [--] [<path>…]

This form is to compare the given two paths on the filesystem. You can omit the `--no-index` option when running the command in a working tree controlled by Git and at least one of the paths points outside the working tree, or when running the command outside a working tree controlled by Git.

git diff [--options] --cached [<commit>] [--] [<path>…]

This form is to view the changes you staged for the next commit relative to the named <commit>. Typically you would want comparison with the latest commit, so if you do not give <commit>, it defaults to HEAD. If HEAD does not exist (e.g. unborn branches) and <commit> is not given, it shows all staged changes. --staged is a synonym of --cached.

git diff [--options] <commit> [--] [<path>…]

This form is to view the changes you have in your working tree relative to the named <commit>. You can use HEAD to compare it with the latest commit, or a branch name to compare with the tip of a different branch.

git diff [--options] <commit> <commit> [--] [<path>…]

This is to view the changes between two arbitrary <commit>.

git diff [--options] <commit>..<commit> [--] [<path>…]

This is synonymous to the previous form. If <commit> on one side is omitted, it will have the same effect as using HEAD instead.

git diff [--options] <commit>...<commit> [--] [<path>…]

This form is to view the changes on the branch containing and up to the second <commit>, starting at a common ancestor of both <commit>. "git diff A...B" is equivalent to "git diff $(git-merge-base A B) B". You can omit any one of <commit>, which has the same effect as using HEAD instead.

Just in case if you are doing something exotic, it should be noted that all of the <commit> in the above description, except in the last two forms that use ".." notations, can be any <tree>.

For a more complete list of ways to spell <commit>, see "SPECIFYING REVISIONS" section in gitrevisions[7]. However, "diff" is about comparing two *endpoints*, not ranges, and the range notations ("<commit>..<commit>" and "<commit>...<commit>") do not mean a range as defined in the "SPECIFYING RANGES" section in gitrevisions[7].

git diff [options] <blob> <blob>

This form is to view the differences between the raw contents of two blob objects.

OPTIONS

-p

-u

--patch

Generate patch (see section on generating patches). This is the default.

-s

--no-patch

Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw

Generate the diff in raw format.

--patch-with-raw

Synonym for `-p --raw`.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

`default`, `myers`

The basic greedy diff algorithm. Currently, this is the default.

`minimal`

Spend extra time to make sure the smallest possible diff is produced.

`patience`

Use "patience diff" algorithm when generating patches.

`histogram`

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured diff.algorithm variable to a non-default value and want to use the default one, then you have to use `--diff-algorithm=default` option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by `<width>`. The width of the filename part can be limited by giving another width `<name-width>` after a comma. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (does not affect `git format-patch`). By giving a third parameter `<count>`, you can limit the output to the first `<count>` lines, followed by `...` if there are more.

These parameters can also be set individually with `--stat-width=<width>`, `--stat-name-width=<name-width>` and `--stat-count=<count>`.

--numstat

Similar to `--stat`, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two `-` instead of saying `0 0`.

--shortstat

Output only the last line of the `--stat` format containing total number of modified files, as well as number of added and deleted lines.

--dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of `--dirstat` can be customized by passing it a comma separated list of parameters. The defaults are controlled by the `diff.dirstat` configuration variable (see git-config[1]). The following parameters are available:

`changes`

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

`lines`

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive `--dirstat` behavior than the

`changes` behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other `--*stat` options.

`files`

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest `--dirstat` behavior, since it does not have to look at the file contents at all.

`cumulative`

Count changes in a child directory for the parent directory as well. Note that when using `cumulative`, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the `noncumulative` parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: `--dirstat=files,10,cumulative`.

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for `-p --stat`.

-z

When `--raw`, `--numstat`, `--name-only` or `--name-status` has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only

Show only names of changed files.

--name-status

Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]

Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule[1] `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--color[=<when>]

Show colored diff. `--color` (i.e. without =<*when*>) is the same as `--color=always`. <*when*> can be one of `always`, `never`, or `auto`. It can be changed by the `color.ui` and `color.diff` configuration settings.

--no-color

Turn off colored diff. This can be used to override configuration settings. It is the same as `--color=never`.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see `--word-diff-regex` below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies `--color`.

plain

Show words as `[-removed-]` and `{+added+}`. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a `+` / `-` / `` ` character at the beginning of the line and extending to the end of the line. Newl``
~` on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies `--word-diff` unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append `|[^[:space:]]` to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, `--word-diff-regex=.` will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see gitattributes[1] or git-config[1]. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to `--word-diff=color` plus (if a regex was specified) `--word-diff-regex=<regex>`.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--check

Warn if changes introduce whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors on lines specified by <kind> in the color specified by `color.diff.whitespace`. <kind> is a comma separated list of `old`, `new`, `context`. When this option is not given, only whitespace errors in `new` lines are highlighted. E.g. `--ws-error-highlight=new,old` highlights whitespace errors on both deleted and added lines. `all` can be used as a short-hand for `old,new,context`.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to `--full-index`, output a binary diff that can be applied with `git-apply`.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. Non default number of digits can be specified with `--abbrev=<n>`.

-B[<n>][/<m>]

--break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number `m` controls this aspect of the -B option (defaults to 60%). `-B/70%` specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number `n` controls this aspect of the -B option (defaults to 50%). `-B20%` specifies that a change with addition and deletion compared to 20% or more of the file’s size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>]

--find-renames[=<n>]

Detect renames. If `n` is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file’s size). For example, `-M90%` means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn’t changed. Without a `%` sign, the number is to be read as a fraction, with a decimal point before it. I.e., `-M5` becomes 0.5, and is thus the same as `-M50%`. Similarly, `-M05` is the same as `-M5%`. To limit detection to exact renames, use `-M100%`. The default similarity index is 50%.

-C[<n>]

--find-copies[=<n>]

Detect copies as well as renames. See also `--find-copies-harder`. If `n` is specified, it has the same meaning as for `-M<n>`.

--find-copies-harder

For performance reasons, by default, `-C` option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one `-C` option has the same effect.

-D

--irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and `/dev/null`. The resulting patch is not meant to be applied with `patch` or `git apply`; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lack enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with `-B`, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The `-M` and `-C` options require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (`A`), Copied (`C`), Deleted (`D`), Modified (`M`), Renamed (`R`), have their type (i.e. regular file, symlink, submodule, …) changed (`T`), are Unmerged (`U`), are Unknown (`X`), or have had their pairing Broken (`B`). Any combination of the filter characters (including none) can be used. When `*` (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter’s use.

It is useful when you're looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into `-S`, and keep going until you get the very first version of the block.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between `-S<regex> --pickaxe-regex` and `-G<regex>`, consider a commit with the following diff in the same file:

```
+    return !regexec(regexp, two-&gt;ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

See the *pickaxe* entry in gitdiffcore[7] for more information.

--pickaxe-all

When `-S` or `-G` finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to `-S` as an extended POSIX regular expression to match.

-O<orderfile>

Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config[1]). To cancel `diff.orderFile`, use `-O/dev/null`.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument.

-a

--text

Treat all files as text.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b

--ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w

--ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other.

-W

--function-context

Show whole surrounding functions of changes.

--exit-code

Make the program exit with codes similar to diff(1). That is, it exits with 1 if there were differences and 0 means no differences.

--quiet

Disable all output of the program. Implies `--exit-code`.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with gitattributes[5], you need to use this option with git-log[1] and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv

--no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See gitattributes[5] for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for git-diff[1] and git-log[1], but not for git-format-patch[1] or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in git-config[1] or gitmodules[5]. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

For more detailed explanation on these common options, see also gitdiffcore[7].

<path>…

The <paths> parameters, when given, are used to limit the diff to the named paths (you can give directory names and get diff for all files under them).

Raw output format

The raw output format from "git-diff-index", "git-diff-tree", "git-diff-files" and "git diff --raw" are very similar.

These commands all compare two sets of things; what is compared differs:

git-diff-index <tree-ish>

compares the <tree-ish> and the files on the filesystem.

git-diff-index --cached <tree-ish>

compares the <tree-ish> and the index.

git-diff-tree [-r] <tree-ish-1> <tree-ish-2> [<pattern>…]

compares the trees named by the two arguments.

git-diff-files [<pattern>…]

compares the index and the files on the filesystem.

The "git-diff-tree" command begins its output by printing the hash of what is being compared. After that, all the commands print one output line per changed file.

An output line is formatted this way:

```
in-place edit  :100644 100644 bcd1234... 0123456... M file0
copy-edit      :100644 100644 abcd123... 1234567... C68 file1 file2
rename-edit    :100644 100644 abcd123... 1234567... R86 file1 file3
create         :000000 100644 0000000... 1234567... A file4
delete         :100644 000000 1234567... 0000000... D file5
unmerged       :000000 000000 0000000... 0000000... U file6
```

That is, from the left to the right:

1. a colon.
2. mode for "src"; 000000 if creation or unmerged.
3. a space.
4. mode for "dst"; 000000 if deletion or unmerged.
5. a space.
6. sha1 for "src"; 0{40} if creation or unmerged.
7. a space.
8. sha1 for "dst"; 0{40} if creation, unmerged or "look at work tree".

9. a space.
10. status, followed by optional "score" number.
11. a tab or a NUL when *-z* option is used.
12. path for "src"
13. a tab or a NUL when *-z* option is used; only exists for C or R.
14. path for "dst"; only exists for C or R.
15. an LF or a NUL when *-z* option is used, to terminate the record.

Possible status letters are:

- A: addition of a file
- C: copy of a file into a new one
- D: deletion of a file
- M: modification of the contents or mode of a file
- R: renaming of a file
- T: change in the type of the file
- U: file is unmerged (you must complete the merge before it can be committed)
- X: "unknown" change type (most probably a bug, please report it)

Status letters C and R are always followed by a score (denoting the percentage of similarity between the source and target of the move or copy). Status letter M may be followed by a score (denoting the percentage of dissimilarity) for file rewrites.

<sha1> is shown as all 0's if a file is new on the filesystem and it is out of sync with the index.

Example:

```
:100644 100644 5be4a4...... 000000...... M file.c
```

When `-z` option is not used, TAB, LF, and backslash characters in pathnames are represented as `\t`, `\n`, and `\\`, respectively.

diff format for merges

"git-diff-tree", "git-diff-files" and "git-diff --raw" can take *-c* or *--cc* option to generate diff output also for merge commits. The output differs from the format described above in the following way:

1. there is a colon for each parent
2. there are more "src" modes and "src" sha1
3. status is concatenated status characters for each parent
4. no optional "score" number
5. single path, only for "dst"

Example:

```
::100644 100644 100644 fabadb8... cc95eb0... 4866510... MM    describe.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The `a/` and `b/` filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, `/dev/null` is *not* used in place of the `a/` or `b/` filenames.

   When rename/copy is involved, `file1` and `file2` show the name of the source file of the rename/copy and the name of the file that rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

```
old mode &lt;mode&gt;
new mode &lt;mode&gt;
deleted file mode &lt;mode&gt;
new file mode &lt;mode&gt;
copy from &lt;path&gt;
copy to &lt;path&gt;
rename from &lt;path&gt;
rename to &lt;path&gt;
similarity index &lt;number&gt;
dissimilarity index &lt;number&gt;
index &lt;hash&gt;..&lt;hash&gt; &lt;mode&gt;
```

File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the `a/` and `b/` prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the SHA-1 checksum before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

4. All the `file1` files in the output refer to files before the commit, and all the `file2` files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

combined diff format

Any diff-generating command can take the `-c` or `--cc` option to produce a *combined diff* when showing a merge. This is the default format when showing merges with git-diff[1] or git-show[1]. Note also that you can give the `-m` option to any of these commands to force generation of diffs with individual parents of a merge.

A *combined diff* format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
    return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +    unsigned char sha1[20];
 +    struct commit *cmit;
    struct commit_list *list;
    static int initialized = 0;
    struct commit_name *n;

 +    if (get_sha1(arg, sha1) < 0)
 +        usage(describe_usage);
 +    cmit = lookup_commit_reference(sha1);
 +    if (!cmit)
 +        usage(describe_usage);
 +
    if (!initialized) {
        initialized = 1;
        for_each_ref(get_name);
```

1. It is preceded with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index &lt;hash&gt;,&lt;hash&gt;..&lt;hash&gt;
   mode &lt;mode&gt;,&lt;mode&gt;..&lt;mode&gt;
   new file mode &lt;mode&gt;
   deleted file mode &lt;mode&gt;,&lt;mode&gt;
   ```

   The `mode <mode>,<mode>..<mode>` line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.

4. Chunk header format is modified to prevent people from accidentally feeding it to `patch -p1`. Combined diff format was created for review of merge commit changes, and was not meant for apply. The change is similar to the change in the extended *index* header:

```
@@@ &lt;from-file-range&gt; &lt;from-file-range&gt; &lt;to-file-range&gt; @@@
```

   There are (number of parents + 1) `@` characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has `-` (minus — appears in A but removed in B), `+` (plus — missing in A but added to B), or `" "` (space — unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A `-` character in the column N means that the line appears in fileN but it does not appear in the result. A `+` character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two `-` removals from both file1 and file2, plus `++` to mean one line that was added does not appear in either file1 or file2). Also eight other lines are the same from file1 but do not appear in file2 (hence prefixed with `+`).

When shown by `git diff-tree -c`, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by `git diff-files -c`, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

other diff formats

The `--summary` option describes newly added, deleted, renamed and copied files. The `--stat` option adds diffstat(1) graph to the output. These options can be combined with other options, such as `-p`, and are meant for human consumption.

When showing a change that involves a rename or a copy, `--stat` output formats the pathnames compactly by combining common prefix and suffix of the pathnames. For example, a change that moves `arch/i386/Makefile` to `arch/x86/Makefile` while modifying 4

lines will be shown like this:

```
arch/{i386 => x86}/Makefile    |   4 +--
```

The `--numstat` option gives the diffstat(1) information but is designed for easier machine consumption. An entry in `--numstat` output looks like this:

```
1    2    README
3    1    arch/{i386 => x86}/Makefile
```

That is, from left to right:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. pathname (possibly with rename/copy information);
6. a newline.

When `-z` output option is in effect, the output is formatted this way:

```
1    2    README NUL
3    1    NUL arch/i386/Makefile NUL arch/x86/Makefile NUL
```

That is:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. a NUL (only exists if renamed/copied);
6. pathname in preimage;
7. a NUL (only exists if renamed/copied);
8. pathname in postimage (only exists if renamed/copied);
9. a NUL.

The extra `NUL` before the preimage path in renamed case is to allow scripts that read the output to tell if the current record being read is a single-path record or a rename/copy record without reading ahead. After reading added and deleted lines, reading up to `NUL` would yield the pathname, but if that is `NUL`, the record will show two paths.

EXAMPLES

Various ways to check your working tree

```
$ git diff            (1)
$ git diff --cached   (2)
$ git diff HEAD       (3)
```

1. Changes in the working tree not yet staged for the next commit.
2. Changes between the index and your last commit; what you would be committing if you run "git commit" without "-a" option.
3. Changes in the working tree since your last commit; what you would be committing if you run "git commit -a"

Comparing with arbitrary commits

```
$ git diff test            (1)
$ git diff HEAD -- ./test  (2)
$ git diff HEAD^ HEAD      (3)
```

1. Instead of using the tip of the current branch, compare with the tip of "test" branch.
2. Instead of comparing with the tip of "test" branch, compare with the tip of the current branch, but limit the comparison to the file "test".
3. Compare the version before the last commit and the last commit.

Comparing branches

```
$ git diff topic master    (1)
$ git diff topic..master   (2)
$ git diff topic...master  (3)
```

1. Changes between the tips of the topic and the master branches.
2. Same as above.
3. Changes that occurred on the master branch since when the topic branch was started off it.

Limiting the diff output

```
$ git diff --diff-filter=MRC            (1)
$ git diff --name-status                (2)
$ git diff arch/i386 include/asm-i386   (3)
```

1. Show only modification, rename, and copy, but not addition or deletion.
2. Show only names and the nature of change, but not actual diff output.
3. Limit diff output to named subtrees.

Munging the diff output

```
$ git diff --find-copies-harder -B -C  (1)
$ git diff -R                          (2)
```

1. Spend extra cycles to find renames, copies and complete rewrites (very expensive).
2. Output diff in reverse.

SEE ALSO

diff(1), git-difftool[1], git-log[1], gitdiffcore[7], git-format-patch[1], git-apply[1]

GIT

Part of the git[1] suite

commit

NAME

git-commit - Record changes to the repository

SYNOPSIS

```
git commit [-a | --interactive | --patch] [-s] [-v] [-u<mode>] [--amend]
       [--dry-run] [(-c | -C | --fixup | --squash) <commit>]
       [-F <file> | -m <msg>] [--reset-author] [--allow-empty]
       [--allow-empty-message] [--no-verify] [-e] [--author=<author>]
       [--date=<date>] [--cleanup=<mode>] [--[no-]status]
       [-i | -o] [-S[<keyid>]] [--] [<file>…]
```

DESCRIPTION

Stores the current contents of the index in a new commit along with a log message from the user describing the changes.

The content to be added can be specified in several ways:

1. by using *git add* to incrementally "add" changes to the index before using the *commit* command (Note: even modified files must be "added");
2. by using *git rm* to remove files from the working tree and the index, again before using the *commit* command;
3. by listing files as arguments to the *commit* command, in which case the commit will ignore changes staged in the index, and instead record the current content of the listed files (which must already be known to Git);
4. by using the -a switch with the *commit* command to automatically "add" changes from all known files (i.e. all files that are already listed in the index) and to automatically "rm" files in the index that have been removed from the working tree, and then perform the actual commit;
5. by using the --interactive or --patch switches with the *commit* command to decide one by one which files or hunks should be part of the commit, before finalizing the operation. See the “Interactive Mode” section of git-add[1] to learn how to operate these modes.

The `--dry-run` option can be used to obtain a summary of what is included by any of the above for the next commit by giving the same set of parameters (options and paths).

If you make a commit and then find a mistake immediately after that, you can recover from it with *git reset*.

OPTIONS

-a

--all

Tell the command to automatically stage files that have been modified and deleted, but new files you have not told Git about are not affected.

-p

--patch

Use the interactive patch selection interface to chose which changes to commit. See git-add[1] for details.

-C <commit>

--reuse-message=<commit>

Take an existing commit object, and reuse the log message and the authorship information (including the timestamp) when creating the commit.

-c <commit>

--reedit-message=<commit>

Like *-C*, but with *-c* the editor is invoked, so that the user can further edit the commit message.

--fixup=<commit>

Construct a commit message for use with `rebase --autosquash`. The commit message will be the subject line from the specified commit with a prefix of "fixup! ". See git-rebase[1] for details.

--squash=<commit>

Construct a commit message for use with `rebase --autosquash`. The commit message subject line is taken from the specified commit with a prefix of "squash! ". Can be used with additional commit message options (`-m` / `-c` / `-C` / `-F`). See git-rebase[1] for details.

--reset-author

When used with -C/-c/--amend options, or when committing after a a conflicting cherry-pick, declare that the authorship of the resulting commit now belongs to the committer. This also renews the author timestamp.

--short

When doing a dry-run, give the output in the short-format. See git-status[1] for details. Implies `--dry-run`.

--branch

Show the branch and tracking info even in short-format.

--porcelain

When doing a dry-run, give the output in a porcelain-ready format. See git-status[1] for details. Implies `--dry-run`.

--long

When doing a dry-run, give the output in a the long-format. Implies `--dry-run`.

-z

--null

When showing `short` or `porcelain` status output, terminate entries in the status output with NUL, instead of LF. If no format is given, implies the `--porcelain` output format.

-F <file>

--file=<file>

Take the commit message from the given file. Use - to read the message from the standard input.

--author=<author>

Override the commit author. Specify an explicit author using the standard `A U Thor <author@example.com>` format. Otherwise <author> is assumed to be a pattern and is used to search for an existing commit by that author (i.e. rev-list --all -i --author=<author>); the commit author is then copied from the first such commit found.

--date=<date>

Override the author date used in the commit.

-m <msg>

--message=<msg>

Use the given <msg> as the commit message. If multiple `-m` options are given, their values are concatenated as separate paragraphs.

-t <file>

--template=<file>

When editing the commit message, start the editor with the contents in the given file. The `commit.template` configuration variable is often used to give this option implicitly to the command. This mechanism can be used by projects that want to guide participants with some hints on what to write in the message in what order. If the user exits the editor without editing the message, the commit is aborted. This has no effect when a message is given by other means, e.g. with the `-m` or `-F` options.

-s

--signoff

Add Signed-off-by line by the committer at the end of the commit log message. The meaning of a signoff depends on the project, but it typically certifies that committer has the rights to submit this work under the same license and agrees to a Developer Certificate of Origin (see http://developercertificate.org/ for more information).

-n

--no-verify

This option bypasses the pre-commit and commit-msg hooks. See also githooks[5].

--allow-empty

Usually recording a commit that has the exact same tree as its sole parent commit is a mistake, and the command prevents you from making such a commit. This option bypasses the safety, and is primarily for use by foreign SCM interface scripts.

--allow-empty-message

Like --allow-empty this command is primarily for use by foreign SCM interface scripts. It allows you to create a commit with an empty commit message without using plumbing commands like git-commit-tree[1].

--cleanup=<mode>

This option determines how the supplied commit message should be cleaned up before committing. The *<mode>* can be `strip`, `whitespace`, `verbatim`, `scissors` or `default`.

strip

Strip leading and trailing empty lines, trailing whitespace, commentary and collapse consecutive empty lines.

whitespace

Same as `strip` except #commentary is not removed.

verbatim

Do not change the message at all.

scissors

Same as `whitespace` , except that everything from (and including) the line " `# ------------------------ >8 ------------------------` " is truncated if the message is to be edited. " `#` " can be customized with core.commentChar.

default

Same as `strip` if the message is to be edited. Otherwise `whitespace` .

The default can be changed by the *commit.cleanup* configuration variable (see git-config[1]).

-e

--edit

The message taken from file with `-F` , command line with `-m` , and from commit object with `-C` are usually used as the commit log message unmodified. This option lets you further edit the message taken from these sources.

--no-edit

Use the selected commit message without launching an editor. For example, `git commit --amend --no-edit` amends a commit without changing its commit message.

--amend

Replace the tip of the current branch by creating a new commit. The recorded tree is prepared as usual (including the effect of the `-i` and `-o` options and explicit pathspec), and the message from the original commit is used as the starting point, instead of an empty message, when no other message is specified from the command line via options such as `-m` , `-F` , `-c` , etc. The new commit has the same parents and author as the current one (the `--reset-author` option can countermand this).

It is a rough equivalent for:

```
$ git reset --soft HEAD^
$ ... do something else to come up with the right tree ...
$ git commit -c ORIG_HEAD
```

but can be used to amend a merge commit.

You should understand the implications of rewriting history if you amend a commit that has already been published. (See the "RECOVERING FROM UPSTREAM REBASE" section in git-rebase[1].)

--no-post-rewrite

Bypass the post-rewrite hook.

-i

--include

Before making a commit out of staged contents so far, stage the contents of paths given on the command line as well. This is usually not what you want unless you are concluding a conflicted merge.

-o

--only

Make a commit by taking the updated working tree contents of the paths specified on the command line, disregarding any contents that have been staged for other paths. This is the default mode of operation of *git commit* if any paths are given on the command line, in which case this option can be omitted. If this option is specified together with *--amend*, then no paths need to be specified, which can be used to amend the last commit without committing changes that have already been staged.

-u[<mode>]

--untracked-files[=<mode>]

Show untracked files.

The mode parameter is optional (defaults to *all*), and is used to specify the handling of untracked files; when -u is not used, the default is *normal*, i.e. show untracked files and directories.

The possible options are:

- *no* - Show no untracked files
- *normal* - Shows untracked files and directories

- *all* - Also shows individual files in untracked directories.

The default can be changed using the status.showUntrackedFiles configuration variable documented in git-config[1].

-v

--verbose

Show unified diff between the HEAD commit and what would be committed at the bottom of the commit message template to help the user describe the commit by reminding what changes the commit has. Note that this diff output doesn't have its lines prefixed with #. This diff will not be a part of the commit message.

If specified twice, show in addition the unified diff between what would be committed and the worktree files, i.e. the unstaged changes to tracked files.

-q

--quiet

Suppress commit summary message.

--dry-run

Do not create a commit, but show a list of paths that are to be committed, paths with local changes that will be left uncommitted and paths that are untracked.

--status

Include the output of git-status[1] in the commit message template when using an editor to prepare the commit message. Defaults to on, but can be used to override configuration variable commit.status.

--no-status

Do not include the output of git-status[1] in the commit message template when using an editor to prepare the default commit message.

-S[<keyid>]

--gpg-sign[=<keyid>]

GPG-sign commits. The `keyid` argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space.

--no-gpg-sign

Countermand `commit.gpgSign` configuration variable that is set to force each and every commit to be signed.

--

Do not interpret any more arguments as options.

<file>…

When files are given on the command line, the command commits the contents of the named files, without recording the changes already staged. The contents of these files are also staged for the next commit on top of what have been staged before.

DATE FORMATS

The GIT_AUTHOR_DATE, GIT_COMMITTER_DATE environment variables and the `--date` option support the following date formats:

Git internal format

It is `<unix timestamp> <time zone offset>`, where `<unix timestamp>` is the number of seconds since the UNIX epoch. `<time zone offset>` is a positive or negative offset from UTC. For example CET (which is 2 hours ahead UTC) is `+0200`.

RFC 2822

The standard email format as described by RFC 2822, for example `Thu, 07 Apr 2005 22:13:13 +0200`.

ISO 8601

Time and date specified by the ISO 8601 standard, for example `2005-04-07T22:13:13`. The parser accepts a space instead of the `T` character as well.

> Note
>
> In addition, the date part is accepted in the following formats: `YYYY.MM.DD`, `MM/DD/YYYY` and `DD.MM.YYYY`.

EXAMPLES

When recording your own work, the contents of modified files in your working tree are temporarily stored to a staging area called the "index" with *git add*. A file can be reverted back, only in the index but not in the working tree, to that of the last commit with `git reset HEAD -- <file>`, which effectively reverts *git add* and prevents the changes

to this file from participating in the next commit. After building the state to be committed incrementally with these commands, `git commit` (without any pathname parameter) is used to record what has been staged so far. This is the most basic form of the command. An example:

```
$ edit hello.c
$ git rm goodbye.c
$ git add hello.c
$ git commit
```

Instead of staging files after each individual change, you can tell `git commit` to notice the changes to the files whose contents are tracked in your working tree and do corresponding `git add` and `git rm` for you. That is, this example does the same as the earlier example if there is no other change in your working tree:

```
$ edit hello.c
$ rm goodbye.c
$ git commit -a
```

The command `git commit -a` first looks at your working tree, notices that you have modified hello.c and removed goodbye.c, and performs necessary `git add` and `git rm` for you.

After staging changes to many files, you can alter the order the changes are recorded in, by giving pathnames to `git commit`. When pathnames are given, the command makes a commit that only records the changes made to the named paths:

```
$ edit hello.c hello.h
$ git add hello.c hello.h
$ edit Makefile
$ git commit Makefile
```

This makes a commit that records the modification to `Makefile`. The changes staged for `hello.c` and `hello.h` are not included in the resulting commit. However, their changes are not lost — they are still staged and merely held back. After the above sequence, if you do:

```
$ git commit
```

this second commit would record the changes to `hello.c` and `hello.h` as expected.

After a merge (initiated by *git merge* or *git pull*) stops because of conflicts, cleanly merged paths are already staged to be committed for you, and paths that conflicted are left in unmerged state. You would have to first check which paths are conflicting with *git status* and after fixing them manually in your working tree, you would stage the result as usual with *git add*:

```
$ git status | grep unmerged
unmerged: hello.c
$ edit hello.c
$ git add hello.c
```

After resolving conflicts and staging the result, `git ls-files -u` would stop mentioning the conflicted path. When you are done, run `git commit` to finally record the merge:

```
$ git commit
```

As with the case to record your own changes, you can use `-a` option to save typing. One difference is that during a merge resolution, you cannot use `git commit` with pathnames to alter the order the changes are committed, because the merge should be recorded as a single commit. In fact, the command refuses to run when given pathnames (but see `-i` option).

DISCUSSION

Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout Git. For example, git-format-patch[1] turns a commit into email, and it uses the title on the Subject line and the rest of the commit in the body.

Git is to some extent character encoding agnostic.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.

- Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (`.git/config` (see git-config[1]), gitignore[5], gitattributes[5] and gitmodules[5]).

  Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that use legacy extended ASCII encodings. However, repositories created on such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.

- Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but *not* UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

Although we encourage that the commit log messages are encoded in UTF-8, both the core and Git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, Git does not forbid it. However, there are a few things to keep in mind.

1. *git commit* and *git commit-tree* issues a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have i18n.commitencoding in `.git/config` file, like this:

   ```
   [i18n]
       commitencoding = ISO-8859-1
   ```

   Commit objects created with the above setting record the value of `i18n.commitencoding` in its `encoding` header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.

2. *git log*, *git show*, *git blame* and friends look at the `encoding` header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with `i18n.logoutputencoding` in `.git/config` file, like this:

   ```
   [i18n]
       logoutputencoding = ISO-8859-1
   ```

   If you do not have this configuration variable, the value of `i18n.commitencoding` is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

ENVIRONMENT AND CONFIGURATION VARIABLES

The editor used to edit the commit log message will be chosen from the GIT_EDITOR environment variable, the core.editor configuration variable, the VISUAL environment variable, or the EDITOR environment variable (in that order). See git-var[1] for details.

HOOKS

This command can run `commit-msg` , `prepare-commit-msg` , `pre-commit` , and `post-commit` hooks. See githooks[5] for more information.

FILES

`$GIT_DIR/COMMIT_EDITMSG`

This file contains the commit message of a commit in progress. If `git commit` exits due to an error before creating a commit, any commit message that has been provided by the user (e.g., in an editor session) will be available in this file, but will be overwritten by the next invocation of `git commit` .

SEE ALSO

git-add[1], git-rm[1], git-mv[1], git-merge[1], git-commit-tree[1]

GIT

Part of the git[1] suite

reset

NAME

git-reset - Reset current HEAD to the specified state

SYNOPSIS

```
git reset [-q] [<tree-ish>] [--] <paths>…
git reset (--patch | -p) [<tree-ish>] [--] [<paths>…]
git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]
```

DESCRIPTION

In the first and second form, copy entries from <tree-ish> to the index. In the third form, set the current branch head (HEAD) to <commit>, optionally modifying index and working tree to match. The <tree-ish>/<commit> defaults to HEAD in all forms.

git reset [-q] [<tree-ish>] [--] <paths>…

This form resets the index entries for all <paths> to their state at <tree-ish>. (It does not affect the working tree or the current branch.)

This means that `git reset <paths>` is the opposite of `git add <paths>`.

After running `git reset <paths>` to update the index entry, you can use git-checkout[1] to check the contents out of the index to the working tree. Alternatively, using git-checkout[1] and specifying a commit, you can copy the contents of a path out of a commit to the index and to the working tree in one go.

git reset (--patch | -p) [<tree-ish>] [--] [<paths>…]

Interactively select hunks in the difference between the index and <tree-ish> (defaults to HEAD). The chosen hunks are applied in reverse to the index.

This means that `git reset -p` is the opposite of `git add -p`, i.e. you can use it to selectively reset hunks. See the “Interactive Mode” section of git-add[1] to learn how to operate the `--patch` mode.

git reset [<mode>] [<commit>]

This form resets the current branch head to <commit> and possibly updates the index (resetting it to the tree of <commit>) and the working tree depending on <mode>. If <mode> is omitted, defaults to "--mixed". The <mode> must be one of the following:

--soft

Does not touch the index file or the working tree at all (but resets the head to <commit>, just like all modes do). This leaves all your changed files "Changes to be committed", as *git status* would put it.

--mixed

Resets the index but not the working tree (i.e., the changed files are preserved but not marked for commit) and reports what has not been updated. This is the default action.

If `-N` is specified, removed paths are marked as intent-to-add (see git-add[1]).

--hard

Resets the index and working tree. Any changes to tracked files in the working tree since <commit> are discarded.

--merge

Resets the index and updates the files in the working tree that are different between <commit> and HEAD, but keeps those which are different between the index and working tree (i.e. which have changes which have not been added). If a file that is different between <commit> and the index has unstaged changes, reset is aborted.

In other words, --merge does something like a *git read-tree -u -m <commit>*, but carries forward unmerged index entries.

--keep

Resets index entries and updates files in the working tree that are different between <commit> and HEAD. If a file that is different between <commit> and HEAD has local changes, reset is aborted.

If you want to undo a commit other than the latest on a branch, git-revert[1] is your friend.

OPTIONS

-q

--quiet

Be quiet, only report errors.

EXAMPLES

Undo add

```
$ edit                                     (1)
$ git add frotz.c filfre.c
$ mailx                                    (2)
$ git reset                                (3)
$ git pull git://info.example.com/ nitfol  (4)
```

1. You are happily working on something, and find the changes in these files are in good order. You do not want to see them when you run "git diff", because you plan to work on other files and changes with these files are distracting.
2. Somebody asks you to pull, and the changes sounds worthy of merging.
3. However, you already dirtied the index (i.e. your index does not match the HEAD commit). But you know the pull you are going to make does not affect frotz.c or filfre.c, so you revert the index changes for these two files. Your changes in working tree remain there.
4. Then you can pull and merge, leaving frotz.c and filfre.c changes still in the working tree.

Undo a commit and redo

```
$ git commit ...
$ git reset --soft HEAD^      (1)
$ edit                        (2)
$ git commit -a -c ORIG_HEAD  (3)
```

1. This is most often done when you remembered what you just committed is incomplete, or you misspelled your commit message, or both. Leaves working tree as it was before "reset".
2. Make corrections to working tree files.
3. "reset" copies the old head to .git/ORIG_HEAD; redo the commit by starting with its log message. If you do not need to edit the message further, you can give -C option instead.

See also the --amend option to git-commit[1].

Undo a commit, making it a topic branch

```
$ git branch topic/wip     (1)
$ git reset --hard HEAD~3  (2)
$ git checkout topic/wip   (3)
```

1. You have made some commits, but realize they were premature to be in the "master" branch. You want to continue polishing them in a topic branch, so create "topic/wip" branch off of the current HEAD.
2. Rewind the master branch to get rid of those three commits.
3. Switch to "topic/wip" branch and keep working.

Undo commits permanently

```
$ git commit ...
$ git reset --hard HEAD~3   (1)
```

1. The last three commits (HEAD, HEAD^, and HEAD~2) were bad and you do not want to ever see them again. Do **not** do this if you have already given these commits to somebody else. (See the "RECOVERING FROM UPSTREAM REBASE" section in git-rebase[1] for the implications of doing so.)

Undo a merge or pull

```
$ git pull                         (1)
Auto-merging nitfol
CONFLICT (content): Merge conflict in nitfol
Automatic merge failed; fix conflicts and then commit the result.
$ git reset --hard                 (2)
$ git pull . topic/branch          (3)
Updating from 41223... to 13134...
Fast-forward
$ git reset --hard ORIG_HEAD       (4)
```

1. Try to update from the upstream resulted in a lot of conflicts; you were not ready to spend a lot of time merging right now, so you decide to do that later.
2. "pull" has not made merge commit, so "git reset --hard" which is a synonym for "git reset --hard HEAD" clears the mess from the index file and the working tree.
3. Merge a topic branch into the current branch, which resulted in a fast-forward.
4. But you decided that the topic branch is not ready for public consumption yet. "pull" or "merge" always leaves the original tip of the current branch in ORIG_HEAD, so resetting hard to it brings your index file and the working tree back to that state, and resets the tip of the branch to that commit.

Undo a merge or pull inside a dirty working tree

```
$ git pull                         (1)
Auto-merging nitfol
Merge made by recursive.
 nitfol                |   20 +++++----
 ...
$ git reset --merge ORIG_HEAD      (2)
```

1. Even if you may have local modifications in your working tree, you can safely say "git pull" when you know that the change in the other branch does not overlap with them.

2. After inspecting the result of the merge, you may find that the change in the other branch is unsatisfactory. Running "git reset --hard ORIG_HEAD" will let you go back to where you were, but it will discard your local changes, which you do not want. "git reset --merge" keeps your local changes.

Interrupted workflow

Suppose you are interrupted by an urgent fix request while you are in the middle of a large change. The files in your working tree are not in any shape to be committed yet, but you need to get to the other branch for a quick bugfix.

```
$ git checkout feature ;# you were working in "feature" branch and
$ work work work       ;# got interrupted
$ git commit -a -m "snapshot WIP"                 (1)
$ git checkout master
$ fix fix fix
$ git commit ;# commit with real log
$ git checkout feature
$ git reset --soft HEAD^ ;# go back to WIP state  (2)
$ git reset                                       (3)
```

1. This commit will get blown away so a throw-away log message is OK.

2. This removes the *WIP* commit from the commit history, and sets your working tree to the state just before you made that snapshot.

3. At this point the index file still has all the WIP changes you committed as *snapshot WIP*. This updates the index to show your WIP files as uncommitted.

See also git-stash[1].

Reset a single file in the index

Suppose you have added a file to your index, but later decide you do not want to add it to your commit. You can remove the file from the index while keeping your changes with git reset.

```
$ git reset -- frotz.c                      (1)
$ git commit -m "Commit files in index"     (2)
$ git add frotz.c                           (3)
```

1. This removes the file from the index while keeping it in the working directory.
2. This commits all other changes in the index.
3. Adds the file to the index again.

Keep changes in working tree while discarding some previous commits

Suppose you are working on something and you commit it, and then you continue working a bit more, but now you think that what you have in your working tree should be in another branch that has nothing to do with what you committed previously. You can start a new branch and reset it while keeping the changes in your working tree.

```
$ git tag start
$ git checkout -b branch1
$ edit
$ git commit ...                        (1)
$ edit
$ git checkout -b branch2               (2)
$ git reset --keep start                (3)
```

1. This commits your first edits in branch1.
2. In the ideal world, you could have realized that the earlier commit did not belong to the new topic when you created and switched to branch2 (i.e. "git checkout -b branch2 start"), but nobody is perfect.
3. But you can use "reset --keep" to remove the unwanted commit after you switched to "branch2".

DISCUSSION

The tables below show what happens when running:

```
git reset --option target
```

to reset the HEAD to another commit (`target`) with the different reset options depending on the state of the files.

In these tables, A, B, C and D are some different states of a file. For example, the first line of the first table means that if a file is in state A in the working tree, in state B in the index, in state C in HEAD and in state D in the target, then "git reset --soft target" will leave the file in the working tree in state A and in the index in state B. It resets (i.e. moves) the HEAD (i.e. the tip of the current branch, if you are on one) to "target" (which has the file in state D).

```
working index HEAD target         working index HEAD
----------------------------------------------------
 A       B     C    D     --soft   A       B     D
          --mixed  A       D     D
          --hard   D       D     D
          --merge (disallowed)
          --keep  (disallowed)
```

```
working index HEAD target         working index HEAD
----------------------------------------------------
 A       B     C    C     --soft   A       B     C
          --mixed  A       C     C
          --hard   C       C     C
          --merge (disallowed)
          --keep   A       C     C
```

```
working index HEAD target         working index HEAD
----------------------------------------------------
 B       B     C    D     --soft   B       B     D
          --mixed  B       D     D
          --hard   D       D     D
          --merge  D       D     D
          --keep  (disallowed)
```

```
working index HEAD target         working index HEAD
----------------------------------------------------
 B       B     C    C     --soft   B       B     C
          --mixed  B       C     C
          --hard   C       C     C
          --merge  C       C     C
          --keep   B       C     C
```

```
working index HEAD target         working index HEAD
----------------------------------------------------
 B       C     C    D     --soft   B       C     D
          --mixed  B       D     D
          --hard   D       D     D
          --merge (disallowed)
          --keep  (disallowed)
```

```
working index HEAD target         working index HEAD
----------------------------------------------------
 B       C     C    C     --soft   B       C     C
          --mixed  B       C     C
          --hard   C       C     C
          --merge  B       C     C
          --keep   B       C     C
```

"reset --merge" is meant to be used when resetting out of a conflicted merge. Any mergy operation guarantees that the working tree file that is involved in the merge does not have local change wrt the index before it starts, and that it writes the result out to the working tree.

So if we see some difference between the index and the target and also between the index and the working tree, then it means that we are not resetting out from a state that a mergy operation left after failing with a conflict. That is why we disallow --merge option in this case.

"reset --keep" is meant to be used when removing some of the last commits in the current branch while keeping changes in the working tree. If there could be conflicts between the changes in the commit we want to remove and the changes in the working tree we want to keep, the reset is disallowed. That’s why it is disallowed if there are both changes between the working tree and HEAD, and between HEAD and the target. To be safe, it is also disallowed when there are unmerged entries.

The following tables show what happens when there are unmerged entries:

```
working index HEAD target         working index HEAD
----------------------------------------------------
 X       U     A    B     --soft  (disallowed)
          --mixed  X       B     B
          --hard   B       B     B
          --merge  B       B     B
          --keep  (disallowed)
```

```
working index HEAD target         working index HEAD
----------------------------------------------------
 X       U     A    A     --soft  (disallowed)
          --mixed  X       A     A
          --hard   A       A     A
          --merge  A       A     A
          --keep  (disallowed)
```

X means any state and U means an unmerged index.

GIT

Part of the git[1] suite

rm

NAME

git-rm - Remove files from the working tree and from the index

SYNOPSIS

```
git rm [-f | --force] [-n] [-r] [--cached] [--ignore-unmatch] [--quiet] [--] <file>…
```

DESCRIPTION

Remove files from the index, or from the working tree and the index. `git rm` will not remove a file from just your working directory. (There is no option to remove a file only from the working tree and yet keep it in the index; use `/bin/rm` if you want to do that.) The files being removed have to be identical to the tip of the branch, and no updates to their contents can be staged in the index, though that default behavior can be overridden with the `-f` option. When `--cached` is given, the staged content has to match either the tip of the branch or the file on disk, allowing the file to be removed from just the index.

OPTIONS

<file>…

Files to remove. Fileglobs (e.g. `*.c`) can be given to remove all matching files. If you want Git to expand file glob characters, you may need to shell-escape them. A leading directory name (e.g. `dir` to remove `dir/file1` and `dir/file2`) can be given to remove all files in the directory, and recursively all sub-directories, but this requires the `-r` option to be explicitly given.

-f

--force

Override the up-to-date check.

-n

--dry-run

Don't actually remove any file(s). Instead, just show if they exist in the index and would otherwise be removed by the command.

-r

Allow recursive removal when a leading directory name is given.

--

This option can be used to separate command-line options from the list of files, (useful when filenames might be mistaken for command-line options).

--cached

Use this option to unstage and remove paths only from the index. Working tree files, whether modified or not, will be left alone.

--ignore-unmatch

Exit with a zero status even if no files matched.

-q

--quiet

`git rm` normally outputs one line (in the form of an `rm` command) for each file removed. This option suppresses that output.

DISCUSSION

The <file> list given to the command can be exact pathnames, file glob patterns, or leading directory names. The command removes only the paths that are known to Git. Giving the name of a file that you have not told Git about does not remove that file.

File globbing matches across directory boundaries. Thus, given two directories `d` and `d2`, there is a difference between using `git rm 'd*'` and `git rm 'd/*'`, as the former will also remove all of directory `d2`.

REMOVING FILES THAT HAVE DISAPPEARED FROM THE FILESYSTEM

There is no option for `git rm` to remove from the index only the paths that have disappeared from the filesystem. However, depending on the use case, there are several ways that can be done.

Using “git commit -a”

If you intend that your next commit should record all modifications of tracked files in the working tree and record all removals of files that have been removed from the working tree with `rm` (as opposed to `git rm`), use `git commit -a`, as it will automatically notice and record all removals. You can also have a similar effect without committing by using `git add -u`.

Using “git add -A”

When accepting a new code drop for a vendor branch, you probably want to record both the removal of paths and additions of new paths as well as modifications of existing paths.

Typically you would first remove all tracked files from the working tree using this command:

```
git ls-files -z | xargs -0 rm -f
```

and then untar the new code in the working tree. Alternately you could *rsync* the changes into the working tree.

After that, the easiest way to record all removals, additions, and modifications in the working tree is:

```
git add -A
```

See git-add[1].

Other ways

If all you really want to do is to remove from the index the files that are no longer present in the working tree (perhaps because your working tree is dirty so that you cannot use `git commit -a`), use the following command:

```
git diff --name-only --diff-filter=D -z | xargs -0 git rm --cached
```

SUBMODULES

Only submodules using a gitfile (which means they were cloned with a Git version 1.7.8 or newer) will be removed from the work tree, as their repository lives inside the .git directory of the superproject. If a submodule (or one of those nested inside it) still uses a .git directory,

`git rm` will fail - no matter if forced or not - to protect the submodule's history. If it exists the submodule.<name> section in the gitmodules[5] file will also be removed and that file will be staged (unless --cached or -n are used).

A submodule is considered up-to-date when the HEAD is the same as recorded in the index, no tracked files are modified and no untracked files that aren't ignored are present in the submodules work tree. Ignored files are deemed expendable and won't stop a submodule's work tree from being removed.

If you only want to remove the local checkout of a submodule from your work tree without committing the removal, use git-submodule[1] `deinit` instead.

EXAMPLES

`git rm Documentation/\*.txt`

Removes all `*.txt` files from the index that are under the `Documentation` directory and any of its subdirectories.

Note that the asterisk `*` is quoted from the shell in this example; this lets Git, and not the shell, expand the pathnames of files and subdirectories under the `Documentation/` directory.

`git rm -f git-*.sh`

Because this example lets the shell expand the asterisk (i.e. you are listing the files explicitly), it does not remove `subdir/git-foo.sh`.

BUGS

Each time a superproject update removes a populated submodule (e.g. when switching between commits before and after the removal) a stale submodule checkout will remain in the old location. Removing the old directory is only safe when it uses a gitfile, as otherwise the history of the submodule will be deleted too. This step will be obsolete when recursive submodule update has been implemented.

SEE ALSO

git-add[1]

GIT

Part of the git[1] suite

mv

NAME

git-mv - Move or rename a file, a directory, or a symlink

SYNOPSIS

```
git mv <options>… <args>…
```

DESCRIPTION

Move or rename a file, directory or symlink.

```
git mv [-v] [-f] [-n] [-k] <source> <destination>
git mv [-v] [-f] [-n] [-k] <source> ... <destination directory>
```

In the first form, it renames <source>, which must exist and be either a file, symlink or directory, to <destination>. In the second form, the last argument has to be an existing directory; the given sources will be moved into this directory.

The index is updated after successful completion, but the change must still be committed.

OPTIONS

-f

--force

Force renaming or moving of a file even if the target exists

-k

Skip move or rename actions which would lead to an error condition. An error happens when a source is neither existing nor controlled by Git, or when it would overwrite an existing file unless *-f* is given.

-n

--dry-run

Do nothing; only show what would happen

-v

--verbose

Report the names of files as they are moved.

SUBMODULES

Moving a submodule using a gitfile (which means they were cloned with a Git version 1.7.8 or newer) will update the gitfile and core.worktree setting to make the submodule work in the new location. It also will attempt to update the submodule.<name>.path setting in the gitmodules[5] file and stage that file (unless -n is used).

BUGS

Each time a superproject update moves a populated submodule (e.g. when switching between commits before and after the move) a stale submodule checkout will remain in the old location and an empty directory will appear in the new location. To populate the submodule again in the new location the user will have to run "git submodule update" afterwards. Removing the old directory is only safe when it uses a gitfile, as otherwise the history of the submodule will be deleted too. Both steps will be obsolete when recursive submodule update has been implemented.

GIT

Part of the git[1] suite

Branching and Merging

branch

NAME

git-branch - List, create, or delete branches

SYNOPSIS

```
git branch [--color[=<when>] | --no-color] [-r | -a]
    [--list] [-v [--abbrev=<length> | --no-abbrev]]
    [--column[=<options>] | --no-column]
    [(--merged | --no-merged | --contains) [<commit>]] [--sort=<key>]
    [--points-at <object>] [<pattern>…]
git branch [--set-upstream | --track | --no-track] [-l] [-f] <branchname> [<start-point>]
git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]
git branch --unset-upstream [<branchname>]
git branch (-m | -M) [<oldbranch>] <newbranch>
git branch (-d | -D) [-r] <branchname>…
git branch --edit-description [<branchname>]
```

DESCRIPTION

If `--list` is given, or if there are no non-option arguments, existing branches are listed; the current branch will be highlighted with an asterisk. Option `-r` causes the remote-tracking branches to be listed, and option `-a` shows both local and remote branches. If a `<pattern>` is given, it is used as a shell wildcard to restrict the output to matching branches. If multiple patterns are given, a branch is shown if it matches any of the patterns. Note that when providing a `<pattern>`, you must use `--list`; otherwise the command is interpreted as branch creation.

With `--contains`, shows only the branches that contain the named commit (in other words, the branches whose tip commits are descendants of the named commit). With `--merged`, only branches merged into the named commit (i.e. the branches whose tip commits are reachable from the named commit) will be listed. With `--no-merged` only branches not merged into the named commit will be listed. If the <commit> argument is missing it defaults to *HEAD* (i.e. the tip of the current branch).

The command's second form creates a new branch head named <branchname> which points to the current *HEAD*, or <start-point> if given.

Note that this will create the new branch, but it will not switch the working tree to it; use "git checkout <newbranch>" to switch to the new branch.

When a local branch is started off a remote-tracking branch, Git sets up the branch (specifically the `branch.<name>.remote` and `branch.<name>.merge` configuration entries) so that *git pull* will appropriately merge from the remote-tracking branch. This behavior may be changed via the global `branch.autoSetupMerge` configuration flag. That setting can be overridden by using the `--track` and `--no-track` options, and changed later using `git branch --set-upstream-to`.

With a `-m` or `-M` option, <oldbranch> will be renamed to <newbranch>. If <oldbranch> had a corresponding reflog, it is renamed to match <newbranch>, and a reflog entry is created to remember the branch renaming. If <newbranch> exists, -M must be used to force the rename to happen.

With a `-d` or `-D` option, `<branchname>` will be deleted. You may specify more than one branch for deletion. If the branch currently has a reflog then the reflog will also be deleted.

Use `-r` together with `-d` to delete remote-tracking branches. Note, that it only makes sense to delete remote-tracking branches if they no longer exist in the remote repository or if *git fetch* was configured not to fetch them again. See also the *prune* subcommand of git-remote[1] for a way to clean up all obsolete remote-tracking branches.

OPTIONS

-d

--delete

Delete a branch. The branch must be fully merged in its upstream branch, or in `HEAD` if no upstream was set with `--track` or `--set-upstream`.

-D

Shortcut for `--delete --force`.

-l

--create-reflog

Create the branch's reflog. This activates recording of all changes made to the branch ref, enabling use of date based sha1 expressions such as "<branchname>@{yesterday}". Note that in non-bare repositories, reflogs are usually enabled by default by the `core.logallrefupdates` config option.

-f

--force

Reset <branchname> to <startpoint> if <branchname> exists already. Without `-f` *git branch* refuses to change an existing branch. In combination with `-d` (or `--delete`), allow deleting the branch irrespective of its merged status. In combination with `-m` (or `--move`), allow renaming the branch even if the new branch name already exists.

-m

--move

Move/rename a branch and the corresponding reflog.

-M

Shortcut for `--move --force`.

--color[=<when>]

Color branches to highlight current, local, and remote-tracking branches. The value must be always (the default), never, or auto.

--no-color

Turn off branch colors, even when the configuration file gives the default to color output. Same as `--color=never`.

--column[=<options>]

--no-column

Display branch listing in columns. See configuration variable column.branch for option syntax. `--column` and `--no-column` without options are equivalent to *always* and *never* respectively.

This option is only applicable in non-verbose mode.

-r

--remotes

List or delete (if used with -d) the remote-tracking branches.

-a

--all

List both remote-tracking branches and local branches.

--list

Activate the list mode. `git branch <pattern>` would try to create a branch, use `git branch --list <pattern>` to list matching branches.

-v

-vv

--verbose

When in list mode, show sha1 and commit subject line for each head, along with relationship to upstream branch (if any). If given twice, print the name of the upstream branch, as well (see also `git remote show <remote>`).

-q

--quiet

Be more quiet when creating or deleting a branch, suppressing non-error messages.

--abbrev=<length>

Alter the sha1's minimum display length in the output listing. The default value is 7 and can be overridden by the `core.abbrev` config option.

--no-abbrev

Display the full sha1s in the output listing rather than abbreviating them.

-t

--track

When creating a new branch, set up `branch.<name>.remote` and `branch.<name>.merge` configuration entries to mark the start-point branch as "upstream" from the new branch. This configuration will tell git to show the relationship between the two branches in `git status` and `git branch -v`. Furthermore, it directs `git pull` without arguments to pull from the upstream when the new branch is checked out.

This behavior is the default when the start point is a remote-tracking branch. Set the branch.autoSetupMerge configuration variable to `false` if you want `git checkout` and `git branch` to always behave as if *--no-track* were given. Set it to `always` if you want this behavior when the start-point is either a local or remote-tracking branch.

--no-track

Do not set up "upstream" configuration, even if the branch.autoSetupMerge configuration variable is true.

--set-upstream

If specified branch does not exist yet or if `--force` has been given, acts exactly like `--track`. Otherwise sets up configuration like `--track` would when creating the branch, except that where branch points to is not changed.

-u <upstream>

--set-upstream-to=<upstream>

Set up <branchname>'s tracking information so <upstream> is considered <branchname>'s upstream branch. If no <branchname> is specified, then it defaults to the current branch.

--unset-upstream

Remove the upstream information for <branchname>. If no branch is specified it defaults to the current branch.

--edit-description

Open an editor and edit the text to explain what the branch is for, to be used by various other commands (e.g. `format-patch`, `request-pull`, and `merge` (if enabled)). Multi-line explanations may be used.

--contains [<commit>]

Only list branches which contain the specified commit (HEAD if not specified). Implies `--list`.

--merged [<commit>]

Only list branches whose tips are reachable from the specified commit (HEAD if not specified). Implies `--list`.

--no-merged [<commit>]

Only list branches whose tips are not reachable from the specified commit (HEAD if not specified). Implies `--list`.

<branchname>

The name of the branch to create or delete. The new branch name must pass all checks defined by git-check-ref-format[1]. Some of these checks may restrict the characters allowed in a branch name.

<start-point>

The new branch head will point to this commit. It may be given as a branch name, a commit-id, or a tag. If this option is omitted, the current HEAD will be used instead.

<oldbranch>

The name of an existing branch to rename.

<newbranch>

The new name for an existing branch. The same restrictions as for <branchname> apply.

--sort=<key>

Sort based on the key given. Prefix `-` to sort in descending order of the value. You may use the --sort=<key> option multiple times, in which case the last key becomes the primary key. The keys supported are the same as those in `git for-each-ref`. Sort order defaults to sorting based on the full refname (including `refs/...` prefix). This lists detached HEAD (if present) first, then local branches and finally remote-tracking branches.

--points-at <object>

Only list branches of the given object.

Examples

Start development from a known tag

```
$ git clone git://git.kernel.org/pub/scm/.../linux-2.6 my2.6
$ cd my2.6
$ git branch my2.6.14 v2.6.14   (1)
$ git checkout my2.6.14
```

1. This step and the next one could be combined into a single step with "checkout -b my2.6.14 v2.6.14".

Delete an unneeded branch

```
$ git clone git://git.kernel.org/.../git.git my.git
$ cd my.git
$ git branch -d -r origin/todo origin/html origin/man   (1)
$ git branch -D test                                    (2)
```

1. Delete the remote-tracking branches "todo", "html" and "man". The next *fetch* or *pull* will create them again unless you configure them not to. See git-fetch[1].

2. Delete the "test" branch even if the "master" branch (or whichever branch is currently checked out) does not have all commits from the test branch.

Notes

If you are creating a branch that you want to checkout immediately, it is easier to use the git checkout command with its `-b` option to create a branch and check it out with a single command.

The options `--contains`, `--merged` and `--no-merged` serve three related but different purposes:

- `--contains <commit>` is used to find all branches which will need special attention if <commit> were to be rebased or amended, since those branches contain the specified <commit>.
- `--merged` is used to find all branches which can be safely deleted, since those branches are fully contained by HEAD.
- `--no-merged` is used to find branches which are candidates for merging into HEAD, since those branches are not fully contained by HEAD.

SEE ALSO

git-check-ref-format[1], git-fetch[1], git-remote[1], “Understanding history: What is a branch?” in the Git User’s Manual.

GIT

Part of the git[1] suite

checkout

NAME

git-checkout - Switch branches or restore working tree files

SYNOPSIS

```
git checkout [-q] [-f] [-m] [<branch>]
git checkout [-q] [-f] [-m] --detach [<branch>]
git checkout [-q] [-f] [-m] [--detach] <commit>
git checkout [-q] [-f] [-m] [[-b|-B|--orphan] <new_branch>] [<start_point>]
git checkout [-f|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] [--] <paths>…
git checkout [-p|--patch] [<tree-ish>] [--] [<paths>…]
```

DESCRIPTION

Updates files in the working tree to match the version in the index or the specified tree. If no paths are given, *git checkout* will also update `HEAD` to set the specified branch as the current branch.

git checkout <branch>

To prepare for working on <branch>, switch to it by updating the index and the files in the working tree, and by pointing HEAD at the branch. Local modifications to the files in the working tree are kept, so that they can be committed to the <branch>.

If <branch> is not found but there does exist a tracking branch in exactly one remote (call it <remote>) with a matching name, treat as equivalent to

```
$ git checkout -b &lt;branch&gt; --track &lt;remote&gt;/&lt;branch&gt;
```

You could omit <branch>, in which case the command degenerates to "check out the current branch", which is a glorified no-op with a rather expensive side-effects to show only the tracking information, if exists, for the current branch.

git checkout -b|-B <new_branch> [<start point>]

Specifying `-b` causes a new branch to be created as if git-branch[1] were called and then checked out. In this case you can use the `--track` or `--no-track` options, which will be passed to *git branch*. As a convenience, `--track` without `-b` implies branch creation; see

the description of `--track` below.

If `-B` is given, <new_branch> is created if it doesn’t exist; otherwise, it is reset. This is the transactional equivalent of

```
$ git branch -f &lt;branch&gt; [&lt;start point&gt;]
$ git checkout &lt;branch&gt;
```

that is to say, the branch is not reset/created unless "git checkout" is successful.

git checkout --detach [<branch>]

git checkout [--detach] <commit>

Prepare to work on top of <commit>, by detaching HEAD at it (see "DETACHED HEAD" section), and updating the index and the files in the working tree. Local modifications to the files in the working tree are kept, so that the resulting working tree will be the state recorded in the commit plus the local modifications.

When the <commit> argument is a branch name, the `--detach` option can be used to detach HEAD at the tip of the branch (`git checkout <branch>` would check out that branch without detaching HEAD).

Omitting <branch> detaches HEAD at the tip of the current branch.

git checkout [-p|--patch] [<tree-ish>] [--] <pathspec>…

When <paths> or `--patch` are given, *git checkout* does **not** switch branches. It updates the named paths in the working tree from the index file or from a named <tree-ish> (most often a commit). In this case, the `-b` and `--track` options are meaningless and giving either of them results in an error. The <tree-ish> argument can be used to specify a specific tree-ish (i.e. commit, tag or tree) to update the index for the given paths before updating the working tree.

git checkout with <paths> or `--patch` is used to restore modified or deleted paths to their original contents from the index or replace paths with the contents from a named <tree-ish> (most often a commit-ish).

The index may contain unmerged entries because of a previous failed merge. By default, if you try to check out such an entry from the index, the checkout operation will fail and nothing will be checked out. Using `-f` will ignore these unmerged entries. The contents from a specific side of the merge can be checked out of the index by using `--ours` or `--theirs` . With `-m` , changes made to the working tree file can be discarded to re-create the original conflicted merge result.

OPTIONS

-q

--quiet

Quiet, suppress feedback messages.

--[no-]progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless `--quiet` is specified. This flag enables progress reporting even if not attached to a terminal, regardless of `--quiet`.

-f

--force

When switching branches, proceed even if the index or the working tree differs from HEAD. This is used to throw away local changes.

When checking out paths from the index, do not fail upon unmerged entries; instead, unmerged entries are ignored.

--ours

--theirs

When checking out paths from the index, check out stage #2 (*ours*) or #3 (*theirs*) for unmerged paths.

Note that during `git rebase` and `git pull --rebase`, *ours* and *theirs* may appear swapped; `--ours` gives the version from the branch the changes are rebased onto, while `--theirs` gives the version from the branch that holds your work that is being rebased.

This is because `rebase` is used in a workflow that treats the history at the remote as the shared canonical one, and treats the work done on the branch you are rebasing as the third-party work to be integrated, and you are temporarily assuming the role of the keeper of the canonical history during the rebase. As the keeper of the canonical history, you need to view the history from the remote as `ours` (i.e. "our shared canonical history"), while what you did on your side branch as `theirs` (i.e. "one contributor's work on top of it").

-b <new_branch>

Create a new branch named <new_branch> and start it at <start_point>; see git-branch[1] for details.

-B <new_branch>

Creates the branch <new_branch> and start it at <start_point>; if it already exists, then reset it to <start_point>. This is equivalent to running "git branch" with "-f"; see git-branch[1] for details.

-t

--track

When creating a new branch, set up "upstream" configuration. See "--track" in git-branch[1] for details.

If no *-b* option is given, the name of the new branch will be derived from the remote-tracking branch, by looking at the local part of the refspec configured for the corresponding remote, and then stripping the initial part up to the "*". This would tell us to use "hack" as the local branch when branching off of "origin/hack" (or "remotes/origin/hack", or even "refs/remotes/origin/hack"). If the given name has no slash, or the above guessing results in an empty name, the guessing is aborted. You can explicitly give a name with *-b* in such a case.

--no-track

Do not set up "upstream" configuration, even if the branch.autoSetupMerge configuration variable is true.

-l

Create the new branch's reflog; see git-branch[1] for details.

--detach

Rather than checking out a branch to work on it, check out a commit for inspection and discardable experiments. This is the default behavior of "git checkout <commit>" when <commit> is not a branch name. See the "DETACHED HEAD" section below for details.

--orphan <new_branch>

Create a new *orphan* branch, named <new_branch>, started from <start_point> and switch to it. The first commit made on this new branch will have no parents and it will be the root of a new history totally disconnected from all the other branches and commits.

The index and the working tree are adjusted as if you had previously run "git checkout <start_point>". This allows you to start a new history that records a set of paths similar to <start_point> by easily running "git commit -a" to make the root commit.

This can be useful when you want to publish the tree from a commit without exposing its full history. You might want to do this to publish an open source branch of a project whose current tree is "clean", but whose full history contains proprietary or otherwise encumbered

bits of code.

If you want to start a disconnected history that records a set of paths that is totally different from the one of <start_point>, then you should clear the index and the working tree right after creating the orphan branch by running "git rm -rf ." from the top level of the working tree. Afterwards you will be ready to prepare your new files, repopulating the working tree, by copying them from elsewhere, extracting a tarball, etc.

--ignore-skip-worktree-bits

In sparse checkout mode, `git checkout -- <paths>` would update only entries matched by <paths> and sparse patterns in $GIT_DIR/info/sparse-checkout. This option ignores the sparse patterns and adds back any files in <paths>.

-m

--merge

When switching branches, if you have local modifications to one or more files that are different between the current branch and the branch to which you are switching, the command refuses to switch branches in order to preserve your modifications in context. However, with this option, a three-way merge between the current branch, your working tree contents, and the new branch is done, and you will be on the new branch.

When a merge conflict happens, the index entries for conflicting paths are left unmerged, and you need to resolve the conflicts and mark the resolved paths with `git add` (or `git rm` if the merge should result in deletion of the path).

When checking out paths from the index, this option lets you recreate the conflicted merge in the specified paths.

--conflict=<style>

The same as --merge option above, but changes the way the conflicting hunks are presented, overriding the merge.conflictStyle configuration variable. Possible values are "merge" (default) and "diff3" (in addition to what is shown by "merge" style, shows the original contents).

-p

--patch

Interactively select hunks in the difference between the <tree-ish> (or the index, if unspecified) and the working tree. The chosen hunks are then applied in reverse to the working tree (and if a <tree-ish> was specified, the index).

This means that you can use `git checkout -p` to selectively discard edits from your current working tree. See the “Interactive Mode” section of git-add[1] to learn how to operate the `--patch` mode.

--ignore-other-worktrees

`git checkout` refuses when the wanted ref is already checked out by another worktree. This option makes it check the ref out anyway. In other words, the ref can be held by more than one worktree.

<branch>

Branch to checkout; if it refers to a branch (i.e., a name that, when prepended with "refs/heads/", is a valid ref), then that branch is checked out. Otherwise, if it refers to a valid commit, your HEAD becomes "detached" and you are no longer on any branch (see below for details).

As a special case, the `"@{-N}"` syntax for the N-th last branch/commit checks out branches (instead of detaching). You may also specify `-` which is synonymous with `"@{-1}"`.

As a further special case, you may use `"A...B"` as a shortcut for the merge base of `A` and `B` if there is exactly one merge base. You can leave out at most one of `A` and `B`, in which case it defaults to `HEAD`.

<new_branch>

Name for the new branch.

<start_point>

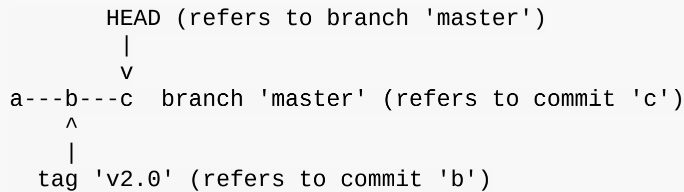
The name of a commit at which to start the new branch; see git-branch[1] for details. Defaults to HEAD.

<tree-ish>

Tree to checkout from (when paths are given). If not specified, the index will be used.

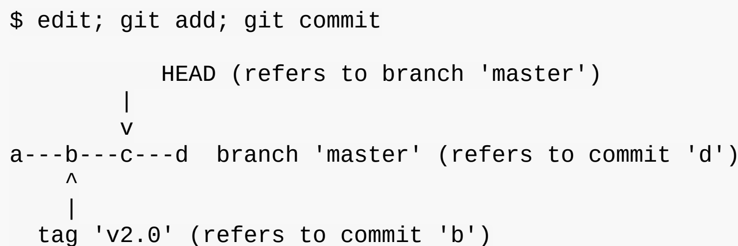
DETACHED HEAD

HEAD normally refers to a named branch (e.g. *master*). Meanwhile, each branch refers to a specific commit. Let’s look at a repo with three commits, one of them tagged, and with branch *master* checked out:



```
       HEAD (refers to branch 'master')
        |
        v
a---b---c  branch 'master' (refers to commit 'c')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

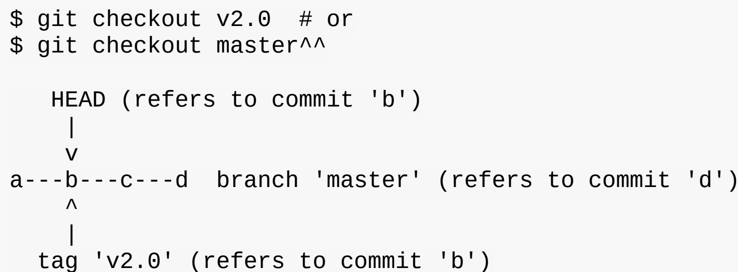
When a commit is created in this state, the branch is updated to refer to the new commit. Specifically, *git commit* creates a new commit *d*, whose parent is commit *c*, and then updates branch *master* to refer to new commit *d*. HEAD still refers to branch *master* and so indirectly now refers to commit *d*:



```
$ edit; git add; git commit

           HEAD (refers to branch 'master')
        |
        v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

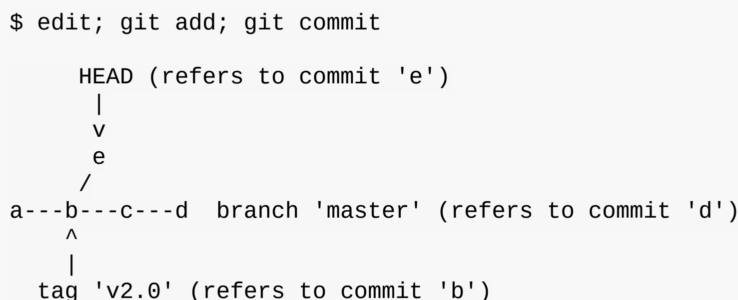
It is sometimes useful to be able to checkout a commit that is not at the tip of any named branch, or even to create a new commit that is not referenced by a named branch. Let's look at what happens when we checkout commit *b* (here we show two ways this may be done):



```
$ git checkout v2.0  # or
$ git checkout master^^

   HEAD (refers to commit 'b')
    |
    v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

Notice that regardless of which checkout command we use, HEAD now refers directly to commit *b*. This is known as being in detached HEAD state. It means simply that HEAD refers to a specific commit, as opposed to referring to a named branch. Let's see what happens when we create a commit:



```
$ edit; git add; git commit

     HEAD (refers to commit 'e')
      |
      v
      e
     /
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

There is now a new commit *e*, but it is referenced only by HEAD. We can of course add yet another commit in this state:

```
$ edit; git add; git commit

     HEAD (refers to commit 'f')
      |
      v
      e---f
     /
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

In fact, we can perform all the normal Git operations. But, let's look at what happens when we then checkout master:

```
$ git checkout master

           HEAD (refers to branch 'master')
      e---f     |
     /          v
a---b---c---d  branch 'master' (refers to commit 'd')
    ^
    |
  tag 'v2.0' (refers to commit 'b')
```

It is important to realize that at this point nothing refers to commit *f*. Eventually commit *f* (and by extension commit *e*) will be deleted by the routine Git garbage collection process, unless we create a reference before that happens. If we have not yet moved away from commit *f*, any of these will create a reference to it:

```
$ git checkout -b foo   (1)
$ git branch foo        (2)
$ git tag foo           (3)
```

1. creates a new branch *foo*, which refers to commit *f*, and then updates HEAD to refer to branch *foo*. In other words, we'll no longer be in detached HEAD state after this command.
2. similarly creates a new branch *foo*, which refers to commit *f*, but leaves HEAD detached.
3. creates a new tag *foo*, which refers to commit *f*, leaving HEAD detached.

If we have moved away from commit *f*, then we must first recover its object name (typically by using git reflog), and then we can create a reference to it. For example, to see the last two commits to which HEAD referred, we can use either of these commands:

```
$ git reflog -2 HEAD # or
$ git log -g -2 HEAD
```

EXAMPLES

1. The following sequence checks out the `master` branch, reverts the `Makefile` to two revisions back, deletes hello.c by mistake, and gets it back from the index.

   ```
   $ git checkout master             (1)
   $ git checkout master~2 Makefile  (2)
   $ rm -f hello.c
   $ git checkout hello.c            (3)
   ```

   i. switch branch

   ii. take a file out of another commit

   iii. restore hello.c from the index

   If you want to check out *all* C source files out of the index, you can say

   ```
   $ git checkout -- '*.c'
   ```

   Note the quotes around `*.c`. The file `hello.c` will also be checked out, even though it is no longer in the working tree, because the file globbing is used to match entries in the index (not in the working tree by the shell).

   If you have an unfortunate branch that is named `hello.c`, this step would be confused as an instruction to switch to that branch. You should instead write:

   ```
   $ git checkout -- hello.c
   ```

2. After working in the wrong branch, switching to the correct branch would be done using:

   ```
   $ git checkout mytopic
   ```

   However, your "wrong" branch and correct "mytopic" branch may differ in files that you have modified locally, in which case the above checkout would fail like this:

   ```
   $ git checkout mytopic
   error: You have local changes to 'frotz'; not switching branches.
   ```

   You can give the `-m` flag to the command, which would try a three-way merge:

```
$ git checkout -m mytopic
Auto-merging frotz
```

After this three-way merge, the local modifications are *not* registered in your index file, so `git diff` would show you what changes you made since the tip of the new branch.

3. When a merge conflict happens during switching branches with the `-m` option, you would see something like this:

```
$ git checkout -m mytopic
Auto-merging frotz
ERROR: Merge conflict in frotz
fatal: merge program failed
```

At this point, `git diff` shows the changes cleanly merged as in the previous example, as well as the changes in the conflicted files. Edit and resolve the conflict and mark it resolved with `git add` as usual:

```
$ edit frotz
$ git add frotz
```

GIT

Part of the git[1] suite

merge

NAME

git-merge - Join two or more development histories together

SYNOPSIS

```
git merge [-n] [--stat] [--no-commit] [--squash] [--[no-]edit]
    [-s <strategy>] [-X <strategy-option>] [-S[<keyid>]]
    [--[no-]rerere-autoupdate] [-m <msg>] [<commit>…]
git merge <msg> HEAD <commit>…
git merge --abort
```

DESCRIPTION

Incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch. This command is used by *git pull* to incorporate changes from another repository and can be used by hand to merge changes from one branch into another.

Assume the following history exists and the current branch is " `master` ":

```
      A---B---C topic
     /
    D---E---F---G master
```

Then " `git merge topic` " will replay the changes made on the `topic` branch since it diverged from `master` (i.e., `E`) until its current commit (`C`) on top of `master` , and record the result in a new commit along with the names of the two parent commits and a log message from the user describing the changes.

```
      A---B---C topic
     /         \
    D---E---F---G---H master
```

The second syntax (<msg> `HEAD` <commit>…) is supported for historical reasons. Do not use it from the command line or in new scripts. It is the same as `git merge -m <msg> <commit>...` .

The third syntax ("`git merge --abort`") can only be run after the merge has resulted in conflicts. *git merge --abort* will abort the merge process and try to reconstruct the pre-merge state. However, if there were uncommitted changes when the merge started (and especially if those changes were further modified after the merge was started), *git merge --abort* will in some cases be unable to reconstruct the original (pre-merge) changes. Therefore:

Warning: Running *git merge* with non-trivial uncommitted changes is discouraged: while possible, it may leave you in a state that is hard to back out of in the case of a conflict.

OPTIONS

--commit

--no-commit

Perform the merge and commit the result. This option can be used to override --no-commit.

With --no-commit perform the merge but pretend the merge failed and do not autocommit, to give the user a chance to inspect and further tweak the merge result before committing.

--edit

-e

--no-edit

Invoke an editor before committing successful mechanical merge to further edit the auto-generated merge message, so that the user can explain and justify the merge. The `--no-edit` option can be used to accept the auto-generated message (this is generally discouraged). The `--edit` (or `-e`) option is still useful if you are giving a draft message with the `-m` option from the command line and want to edit it in the editor.

Older scripts may depend on the historical behaviour of not allowing the user to edit the merge log message. They will see an editor opened when they run `git merge`. To make it easier to adjust such scripts to the updated behaviour, the environment variable `GIT_MERGE_AUTOEDIT` can be set to `no` at the beginning of them.

--ff

When the merge resolves as a fast-forward, only update the branch pointer, without creating a merge commit. This is the default behavior.

--no-ff

Create a merge commit even when the merge resolves as a fast-forward. This is the default behaviour when merging an annotated (and possibly signed) tag.

--ff-only

Refuse to merge and exit with a non-zero status unless the current `HEAD` is already up-to-date or the merge can be resolved as a fast-forward.

--log[=<n>]

--no-log

In addition to branch names, populate the log message with one-line descriptions from at most <n> actual commits that are being merged. See also git-fmt-merge-msg[1].

With --no-log do not list one-line descriptions from the actual commits being merged.

--stat

-n

--no-stat

Show a diffstat at the end of the merge. The diffstat is also controlled by the configuration option merge.stat.

With -n or --no-stat do not show a diffstat at the end of the merge.

--squash

--no-squash

Produce the working tree and index state as if a real merge happened (except for the merge information), but do not actually make a commit, move the `HEAD`, or record `$GIT_DIR/MERGE_HEAD` (to cause the next `git commit` command to create a merge commit). This allows you to create a single commit on top of the current branch whose effect is the same as merging another branch (or more in case of an octopus).

With --no-squash perform the merge and commit the result. This option can be used to override --squash.

-s <strategy>

--strategy=<strategy>

Use the given merge strategy; can be supplied more than once to specify them in the order they should be tried. If there is no `-s` option, a built-in list of strategies is used instead (*git merge-recursive* when merging a single head, *git merge-octopus* otherwise).

-X <option>

--strategy-option=<option>

Pass merge strategy specific option through to the merge strategy.

--verify-signatures

--no-verify-signatures

Verify that the commits being merged have good and trusted GPG signatures and abort the merge in case they do not.

--summary

--no-summary

Synonyms to --stat and --no-stat; these are deprecated and will be removed in the future.

-q

--quiet

Operate quietly. Implies --no-progress.

-v

--verbose

Be verbose.

--progress

--no-progress

Turn progress on/off explicitly. If neither is specified, progress is shown if standard error is connected to a terminal. Note that not all merge strategies may support progress reporting.

-S[<keyid>]

--gpg-sign[=<keyid>]

GPG-sign the resulting merge commit. The `keyid` argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space.

-m <msg>

Set the commit message to be used for the merge commit (in case one is created).

If `--log` is specified, a shortlog of the commits being merged will be appended to the specified message.

The *git fmt-merge-msg* command can be used to give a good default for automated *git merge* invocations. The automated message can include the branch description.

--[no-]rerere-autoupdate

Allow the rerere mechanism to update the index with the result of auto-conflict resolution if possible.

--abort

Abort the current conflict resolution process, and try to reconstruct the pre-merge state.

If there were uncommitted worktree changes present when the merge started, *git merge --abort* will in some cases be unable to reconstruct these changes. It is therefore recommended to always commit or stash your changes before running *git merge*.

git merge --abort is equivalent to *git reset --merge* when `MERGE_HEAD` is present.

<commit>…

Commits, usually other branch heads, to merge into our branch. Specifying more than one commit will create a merge with more than two parents (affectionately called an Octopus merge).

If no commit is given from the command line, merge the remote-tracking branches that the current branch is configured to use as its upstream. See also the configuration section of this manual page.

When `FETCH_HEAD` (and no other commit) is specified, the branches recorded in the `.git/FETCH_HEAD` file by the previous invocation of `git fetch` for merging are merged to the current branch.

PRE-MERGE CHECKS

Before applying outside changes, you should get your own work in good shape and committed locally, so it will not be clobbered if there are conflicts. See also git-stash[1]. *git pull* and *git merge* will stop without doing anything when local uncommitted changes overlap with files that *git pull*/*git merge* may need to update.

To avoid recording unrelated changes in the merge commit, *git pull* and *git merge* will also abort if there are any changes registered in the index relative to the `HEAD` commit. (One exception is when the changed index entries are in the state that would result from the merge already.)

If all named commits are already ancestors of `HEAD`, *git merge* will exit early with the message "Already up-to-date."

FAST-FORWARD MERGE

Often the current branch head is an ancestor of the named commit. This is the most common case especially when invoked from *git pull*: you are tracking an upstream repository, you have committed no local changes, and now you want to update to a newer upstream revision. In this case, a new commit is not needed to store the combined history; instead, the `HEAD` (along with the index) is updated to point at the named commit, without creating an extra merge commit.

This behavior can be suppressed with the `--no-ff` option.

TRUE MERGE

Except in a fast-forward merge (see above), the branches to be merged must be tied together by a merge commit that has both of them as its parents.

A merged version reconciling the changes from all branches to be merged is committed, and your `HEAD`, index, and working tree are updated to it. It is possible to have modifications in the working tree as long as they do not overlap; the update will preserve them.

When it is not obvious how to reconcile the changes, the following happens:

1. The `HEAD` pointer stays the same.
2. The `MERGE_HEAD` ref is set to point to the other branch head.
3. Paths that merged cleanly are updated both in the index file and in your working tree.
4. For conflicting paths, the index file records up to three versions: stage 1 stores the version from the common ancestor, stage 2 from `HEAD`, and stage 3 from `MERGE_HEAD` (you can inspect the stages with `git ls-files -u`). The working tree files contain the result of the "merge" program; i.e. 3-way merge results with familiar conflict markers `<<<` `===` `>>>`.
5. No other changes are made. In particular, the local modifications you had before you started merge will stay the same and the index entries for them stay as they were, i.e. matching `HEAD`.

If you tried a merge which resulted in complex conflicts and want to start over, you can recover with `git merge --abort`.

MERGING TAG

When merging an annotated (and possibly signed) tag, Git always creates a merge commit even if a fast-forward merge is possible, and the commit message template is prepared with the tag message. Additionally, if the tag is signed, the signature check is reported as a comment in the message template. See also git-tag[1].

When you want to just integrate with the work leading to the commit that happens to be tagged, e.g. synchronizing with an upstream release point, you may not want to make an unnecessary merge commit.

In such a case, you can "unwrap" the tag yourself before feeding it to `git merge`, or pass `--ff-only` when you do not have any work on your own. e.g.

```
git fetch origin
git merge v1.2.3^0
git merge --ff-only v1.2.3
```

HOW CONFLICTS ARE PRESENTED

During a merge, the working tree files are updated to reflect the result of the merge. Among the changes made to the common ancestor's version, non-overlapping ones (that is, you changed an area of the file while the other side left that area intact, or vice versa) are incorporated in the final result verbatim. When both sides made changes to the same area, however, Git cannot randomly pick one side over the other, and asks you to resolve it by leaving what both sides did to that area.

By default, Git uses the same style as the one used by the "merge" program from the RCS suite to present such a conflicted hunk, like this:

```
Here are lines that are either unchanged from the common
ancestor, or cleanly resolved because only one side changed.
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified.
```

The area where a pair of conflicting changes happened is marked with markers `<<<<<<<`, `=======`, and `>>>>>>>`. The part before the `=======` is typically your side, and the part afterwards is typically their side.

The default format does not show what the original said in the conflicting area. You cannot tell how many lines are deleted and replaced with Barbie's remark on your side. The only thing you can tell is that your side wants to say it is hard and you'd prefer to go shopping, while the other side wants to claim it is easy.

An alternative style can be used by setting the "merge.conflictStyle" configuration variable to "diff3". In "diff3" style, the above conflict may look like this:

```
Here are lines that are either unchanged from the common
ancestor, or cleanly resolved because only one side changed.
<<<<<<< yours:sample.txt
Conflict resolution is hard;
let's go shopping.
|||||||
Conflict resolution is hard.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified.
```

In addition to the `<<<<<<<`, `=======`, and `>>>>>>>` markers, it uses another `|||||||` marker that is followed by the original text. You can tell that the original just stated a fact, and your side simply gave in to that statement and gave up, while the other side tried to have a more positive attitude. You can sometimes come up with a better resolution by viewing the original.

HOW TO RESOLVE CONFLICTS

After seeing a conflict, you can do two things:

- Decide not to merge. The only clean-ups you need are to reset the index file to the `HEAD` commit to reverse 2. and to clean up working tree changes made by 2. and 3.; `git merge --abort` can be used for this.
- Resolve the conflicts. Git will mark the conflicts in the working tree. Edit the files into shape and *git add* them to the index. Use *git commit* to seal the deal.

You can work through the conflict with a number of tools:

- Use a mergetool. `git mergetool` to launch a graphical mergetool which will work you through the merge.
- Look at the diffs. `git diff` will show a three-way diff, highlighting changes from both the `HEAD` and `MERGE_HEAD` versions.
- Look at the diffs from each branch. `git log --merge -p <path>` will show diffs first for the `HEAD` version and then the `MERGE_HEAD` version.
- Look at the originals. `git show :1:filename` shows the common ancestor, `git show :2:filename` shows the `HEAD` version, and `git show :3:filename` shows the `MERGE_HEAD` version.

EXAMPLES

- Merge branches `fixes` and `enhancements` on top of the current branch, making an octopus merge:

  ```
  $ git merge fixes enhancements
  ```

- Merge branch `obsolete` into the current branch, using `ours` merge strategy:

  ```
  $ git merge -s ours obsolete
  ```

- Merge branch `maint` into the current branch, but do not make a new commit automatically:

  ```
  $ git merge --no-commit maint
  ```

  This can be used when you want to include further changes to the merge, or want to write your own merge commit message.

  You should refrain from abusing this option to sneak substantial changes into a merge commit. Small fixups like bumping release/version name would be acceptable.

MERGE STRATEGIES

The merge mechanism (`git merge` and `git pull` commands) allows the backend *merge strategies* to be chosen with `-s` option. Some strategies can also take their own options, which can be passed by giving `-X<option>` arguments to `git merge` and/or `git pull` .

resolve

This can only resolve two heads (i.e. the current branch and another branch you pulled from) using a 3-way merge algorithm. It tries to carefully detect criss-cross merge ambiguities and is considered generally safe and fast.

recursive

This can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge. This has been reported to result in fewer merge conflicts without causing mismerges by tests done on

actual merge commits taken from Linux 2.6 kernel development history. Additionally this can detect and handle merges involving renames. This is the default merge strategy when pulling or merging one branch.

The *recursive* strategy can take the following options:

ours

This option forces conflicting hunks to be auto-resolved cleanly by favoring *our* version. Changes from the other tree that do not conflict with our side are reflected to the merge result. For a binary file, the entire contents are taken from our side.

This should not be confused with the *ours* merge strategy, which does not even look at what the other tree contains at all. It discards everything the other tree did, declaring *our* history contains all that happened in it.

theirs

This is the opposite of *ours*.

patience

With this option, *merge-recursive* spends a little extra time to avoid mismerges that sometimes occur due to unimportant matching lines (e.g., braces from distinct functions). Use this when the branches to be merged have diverged wildly. See also git-diff[1] `--patience`.

diff-algorithm=[patience|minimal|histogram|myers]

Tells *merge-recursive* to use a different diff algorithm, which can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also git-diff[1] `--diff-algorithm`.

ignore-space-change

ignore-all-space

ignore-space-at-eol

Treats lines with the indicated type of whitespace change as unchanged for the sake of a three-way merge. Whitespace changes mixed with other changes to a line are not ignored. See also git-diff[1] `-b`, `-w`, and `--ignore-space-at-eol`.

- If *their* version only introduces whitespace changes to a line, *our* version is used;
- If *our* version introduces whitespace changes but *their* version includes a substantial change, *their* version is used;

- Otherwise, the merge proceeds in the usual way.

renormalize

This runs a virtual check-out and check-in of all three stages of a file when resolving a three-way merge. This option is meant to be used when merging branches with different clean filters or end-of-line normalization rules. See "Merging branches with differing checkin/checkout attributes" in gitattributes[5] for details.

no-renormalize

Disables the `renormalize` option. This overrides the `merge.renormalize` configuration variable.

no-renames

Turn off rename detection. See also git-diff[1] `--no-renames`.

find-renames[=<n>]

Turn on rename detection, optionally setting the similarity threshold. This is the default. See also git-diff[1] `--find-renames`.

rename-threshold=<n>

Deprecated synonym for `find-renames=<n>`.

subtree[=<path>]

This option is a more advanced form of *subtree* strategy, where the strategy makes a guess on how two trees must be shifted to match with each other when merging. Instead, the specified path is prefixed (or stripped from the beginning) to make the shape of two trees to match.

octopus

This resolves cases with more than two heads, but refuses to do a complex merge that needs manual resolution. It is primarily meant to be used for bundling topic branch heads together. This is the default merge strategy when pulling or merging more than one branch.

ours

This resolves any number of heads, but the resulting tree of the merge is always that of the current branch head, effectively ignoring all changes from all other branches. It is meant to be used to supersede old development history of side branches. Note that this is different from the -Xours option to the *recursive* merge strategy.

subtree

This is a modified recursive strategy. When merging trees A and B, if B corresponds to a subtree of A, B is first adjusted to match the tree structure of A, instead of reading the trees at the same level. This adjustment is also done to the common ancestor tree.

With the strategies that use 3-way merge (including the default, *recursive*), if a change is made on both branches, but later reverted on one of the branches, that change will be present in the merged result; some people find this behavior confusing. It occurs because only the heads and the merge base are considered when performing a merge, not the individual commits. The merge algorithm therefore considers the reverted change as no change at all, and substitutes the changed version instead.

CONFIGURATION

merge.conflictStyle

Specify the style in which conflicted hunks are written out to working tree files upon merge. The default is "merge", which shows a `<<<<<<<` conflict marker, changes made by one side, a `=======` marker, changes made by the other side, and then a `>>>>>>>` marker. An alternate style, "diff3", adds a `|||||||` marker and the original text before the `=======` marker.

merge.defaultToUpstream

If merge is called without any commit argument, merge the upstream branches configured for the current branch by using their last observed values stored in their remote-tracking branches. The values of the `branch.<current branch>.merge` that name the branches at the remote named by `branch.<current branch>.remote` are consulted, and then they are mapped via `remote.<remote>.fetch` to their corresponding remote-tracking branches, and the tips of these tracking branches are merged.

merge.ff

By default, Git does not create an extra merge commit when merging a commit that is a descendant of the current commit. Instead, the tip of the current branch is fast-forwarded. When set to `false`, this variable tells Git to create an extra merge commit in such a case (equivalent to giving the `--no-ff` option from the command line). When set to `only`, only such fast-forward merges are allowed (equivalent to giving the `--ff-only` option from the command line).

fmt-merge-msg-config.txt

merge.renameLimit

The number of files to consider when performing rename detection during a merge; if not specified, defaults to the value of diff.renameLimit.

merge.renormalize

Tell Git that canonical representation of files in the repository has changed over time (e.g. earlier commits record text files with CRLF line endings, but recent ones use LF line endings). In such a repository, Git can convert the data recorded in commits to a canonical form before performing a merge to reduce unnecessary conflicts. For more information, see section "Merging branches with differing checkin/checkout attributes" in gitattributes[5].

merge.stat

Whether to print the diffstat between ORIG_HEAD and the merge result at the end of the merge. True by default.

merge.tool

Controls which merge tool is used by git-mergetool[1]. The list below shows the valid built-in values. Any other value is treated as a custom merge tool and requires that a corresponding mergetool.<tool>.cmd variable is defined.

mergetools-merge.txt

merge.verbosity

Controls the amount of output shown by the recursive merge strategy. Level 0 outputs nothing except a final error message if conflicts were detected. Level 1 outputs only conflicts, 2 outputs conflicts and file changes. Level 5 and above outputs debugging information. The default is level 2. Can be overridden by the *GIT_MERGE_VERBOSITY* environment variable.

merge.<driver>.name

Defines a human-readable name for a custom low-level merge driver. See gitattributes[5] for details.

merge.<driver>.driver

Defines the command that implements a custom low-level merge driver. See gitattributes[5] for details.

merge.<driver>.recursive

Names a low-level merge driver to be used when performing an internal merge between common ancestors. See gitattributes[5] for details.

branch.<name>.mergeOptions

Sets default options for merging into branch <name>. The syntax and supported options are the same as those of *git merge*, but option values containing whitespace characters are currently not supported.

SEE ALSO

git-fmt-merge-msg[1], git-pull[1], gitattributes[5], git-reset[1], git-diff[1], git-ls-files[1], git-add[1], git-rm[1], git-mergetool[1]

GIT

Part of the git[1] suite

mergetool

NAME

git-mergetool - Run merge conflict resolution tools to resolve merge conflicts

SYNOPSIS

```
git mergetool [--tool=<tool>] [-y | --[no-]prompt] [<file>…]
```

DESCRIPTION

Use `git mergetool` to run one of several merge utilities to resolve merge conflicts. It is typically run after *git merge*.

If one or more <file> parameters are given, the merge tool program will be run to resolve differences on each file (skipping those without conflicts). Specifying a directory will include all unresolved files in that path. If no <file> names are specified, *git mergetool* will run the merge tool program on every file with merge conflicts.

OPTIONS

-t <tool>

--tool=<tool>

Use the merge resolution program specified by <tool>. Valid values include emerge, gvimdiff, kdiff3, meld, vimdiff, and tortoisemerge. Run `git mergetool --tool-help` for the list of valid <tool> settings.

If a merge resolution program is not specified, *git mergetool* will use the configuration variable `merge.tool`. If the configuration variable `merge.tool` is not set, *git mergetool* will pick a suitable default.

You can explicitly provide a full path to the tool by setting the configuration variable `mergetool.<tool>.path`. For example, you can configure the absolute path to kdiff3 by setting `mergetool.kdiff3.path`. Otherwise, *git mergetool* assumes the tool is available in PATH.

Instead of running one of the known merge tool programs, *git mergetool* can be customized to run an alternative program by specifying the command line to invoke in a configuration variable `mergetool.<tool>.cmd`.

When *git mergetool* is invoked with this tool (either through the `-t` or `--tool` option or the `merge.tool` configuration variable) the configured command line will be invoked with `$BASE` set to the name of a temporary file containing the common base for the merge, if available; `$LOCAL` set to the name of a temporary file containing the contents of the file on the current branch; `$REMOTE` set to the name of a temporary file containing the contents of the file to be merged, and `$MERGED` set to the name of the file to which the merge tool should write the result of the merge resolution.

If the custom merge tool correctly indicates the success of a merge resolution with its exit code, then the configuration variable `mergetool.<tool>.trustExitCode` can be set to `true`. Otherwise, *git mergetool* will prompt the user to indicate the success of the resolution after the custom tool has exited.

--tool-help

Print a list of merge tools that may be used with `--tool`.

-y

--no-prompt

Don’t prompt before each invocation of the merge resolution program. This is the default if the merge resolution program is explicitly specified with the `--tool` option or with the `merge.tool` configuration variable.

--prompt

Prompt before each invocation of the merge resolution program to give the user a chance to skip the path.

TEMPORARY FILES

`git mergetool` creates `*.orig` backup files while resolving merges. These are safe to remove once a file has been merged and its `git mergetool` session has completed.

Setting the `mergetool.keepBackup` configuration variable to `false` causes `git mergetool` to automatically remove the backup as files are successfully merged.

GIT

Part of the git[1] suite

log

NAME

git-log - Show commit logs

SYNOPSIS

```
git log [<options>] [<revision range>] [[\--] <path>…]
```

DESCRIPTION

Shows the commit logs.

The command takes options applicable to the `git rev-list` command to control what is shown and how, and options applicable to the `git diff-*` commands to control how the changes each commit introduces are shown.

OPTIONS

--follow

Continue listing the history of a file beyond renames (works only for a single file).

--no-decorate

--decorate[=short|full|no]

Print out the ref names of any commits that are shown. If *short* is specified, the ref name prefixes *refs/heads/*, *refs/tags/* and *refs/remotes/* will not be printed. If *full* is specified, the full ref name (including prefix) will be printed. The default option is *short*.

--source

Print out the ref name given on the command line by which each commit was reached.

--use-mailmap

Use mailmap file to map author and committer names and email addresses to canonical real names and email addresses. See git-shortlog[1].

--full-diff

Without this flag, `git log -p <path>...` shows commits that touch the specified paths, and diffs about the same specified paths. With this, the full diff is shown for commits that touch the specified paths; this means that "<path>…" limits only commits, and doesn’t limit diff for those commits.

Note that this affects all diff-based output types, e.g. those produced by `--stat`, etc.

--log-size

Include a line “log size <number>” in the output for each commit, where <number> is the length of that commit’s message in bytes. Intended to speed up tools that read log messages from `git log` output by allowing them to allocate space in advance.

-L <start>,<end>:<file>

-L :<funcname>:<file>

Trace the evolution of the line range given by "<start>,<end>" (or the function name regex <funcname>) within the <file>. You may not give any pathspec limiters. This is currently limited to a walk starting from a single revision, i.e., you may only give zero or one positive revision arguments. You can specify this option more than once.

<start> and <end> can take one of these forms:

- number

  If <start> or <end> is a number, it specifies an absolute line number (lines count from 1).

- /regex/

  This form will use the first line matching the given POSIX regex. If <start> is a regex, it will search from the end of the previous `-L` range, if any, otherwise from the start of file. If <start> is “^/regex/”, it will search from the start of file. If <end> is a regex, it will search starting at the line given by <start>.

- +offset or -offset

  This is only valid for <end> and will specify a number of lines before or after the line given by <start>.

If “:<funcname>” is given in place of <start> and <end>, it is a regular expression that denotes the range from the first funcname line that matches <funcname>, up to the next funcname line. “:<funcname>” searches from the end of the previous `-L` range, if any, otherwise from the start of file. “^:<funcname>” searches from the start of file.

<revision range>

Show only commits in the specified revision range. When no <revision range> is specified, it defaults to `HEAD` (i.e. the whole history leading to the current commit). `origin..HEAD` specifies all the commits reachable from the current commit (i.e. `HEAD`), but not from `origin` . For a complete list of ways to spell <revision range>, see the *Specifying Ranges* section of gitrevisions[7].

[--] <path>…

Show only commits that are enough to explain how the files that match the specified paths came to be. See *History Simplification* below for details and other simplification modes.

Paths may need to be prefixed with ‘`-- ’' to separate them from options or the revision range, when confusion arises.

Commit Limiting

Besides specifying a range of commits that should be listed using the special notations explained in the description, additional commit limiting may be applied.

Using more options generally further limits the output (e.g. `--since=<date1>` limits to commits newer than `<date1>` , and using it with `--grep=<pattern>` further limits to commits whose log message has a line that matches `<pattern>`), unless otherwise noted.

Note that these are applied before commit ordering and formatting options, such as `--reverse` .

-<number>

-n <number>

--max-count=<number>

Limit the number of commits to output.

--skip=<number>

Skip *number* commits before starting to show the commit output.

--since=<date>

--after=<date>

Show commits more recent than a specific date.

--until=<date>

--before=<date>

Show commits older than a specific date.

--author=<pattern>

--committer=<pattern>

Limit the commits output to ones with author/committer header lines that match the specified pattern (regular expression). With more than one `--author=<pattern>`, commits whose author matches any of the given patterns are chosen (similarly for multiple `--committer=<pattern>`).

--grep-reflog=<pattern>

Limit the commits output to ones with reflog entries that match the specified pattern (regular expression). With more than one `--grep-reflog`, commits whose reflog message matches any of the given patterns are chosen. It is an error to use this option unless `--walk-reflogs` is in use.

--grep=<pattern>

Limit the commits output to ones with log message that matches the specified pattern (regular expression). With more than one `--grep=<pattern>`, commits whose message matches any of the given patterns are chosen (but see `--all-match`).

When `--show-notes` is in effect, the message from the notes is matched as if it were part of the log message.

--all-match

Limit the commits output to ones that match all given `--grep`, instead of ones that match at least one.

--invert-grep

Limit the commits output to ones with log message that do not match the pattern specified with `--grep=<pattern>`.

-i

--regexp-ignore-case

Match the regular expression limiting patterns without regard to letter case.

--basic-regexp

Consider the limiting patterns to be basic regular expressions; this is the default.

-E

--extended-regexp

Consider the limiting patterns to be extended regular expressions instead of the default basic regular expressions.

-F

--fixed-strings

Consider the limiting patterns to be fixed strings (don’t interpret pattern as a regular expression).

--perl-regexp

Consider the limiting patterns to be Perl-compatible regular expressions. Requires libpcre to be compiled in.

--remove-empty

Stop when a given path disappears from the tree.

--merges

Print only merge commits. This is exactly the same as `--min-parents=2`.

--no-merges

Do not print commits with more than one parent. This is exactly the same as `--max-parents=1`.

--min-parents=<number>

--max-parents=<number>

--no-min-parents

--no-max-parents

Show only commits which have at least (or at most) that many parent commits. In particular, `--max-parents=1` is the same as `--no-merges`, `--min-parents=2` is the same as `--merges`. `--max-parents=0` gives all root commits and `--min-parents=3` all octopus merges.

`--no-min-parents` and `--no-max-parents` reset these limits (to no limit) again. Equivalent forms are `--min-parents=0` (any commit has 0 or more parents) and `--max-parents=-1` (negative numbers denote no upper limit).

--first-parent

Follow only the first parent commit upon seeing a merge commit. This option can give a better overview when viewing the evolution of a particular topic branch, because merges into a topic branch tend to be only about adjusting to updated upstream from time to time, and this option allows you to ignore the individual commits brought in to your history by such a merge. Cannot be combined with --bisect.

--not

Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next `--not` .

--all

Pretend as if all the refs in `refs/` are listed on the command line as *<commit>*.

--branches[=<pattern>]

Pretend as if all the refs in `refs/heads` are listed on the command line as *<commit>*. If *<pattern>* is given, limit branches to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--tags[=<pattern>]

Pretend as if all the refs in `refs/tags` are listed on the command line as *<commit>*. If *<pattern>* is given, limit tags to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--remotes[=<pattern>]

Pretend as if all the refs in `refs/remotes` are listed on the command line as *<commit>*. If *<pattern>* is given, limit remote-tracking branches to ones matching given shell glob. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--glob=<glob-pattern>

Pretend as if all the refs matching shell glob *<glob-pattern>* are listed on the command line as *<commit>*. Leading *refs/*, is automatically prepended if missing. If pattern lacks *?*, ***, or *[*, */** at the end is implied.

--exclude=<glob-pattern>

Do not include refs matching *<glob-pattern>* that the next `--all` , `--branches` , `--tags` , `--remotes` , or `--glob` would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next `--all` , `--branches` , `--tags` , `--remotes` , or `--glob` option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with `refs/heads`, `refs/tags`, or `refs/remotes` when applied to `--branches`, `--tags`, or `--remotes`, respectively, and they must begin with `refs/` when applied to `--glob` or `--all`. If a trailing /* is intended, it must be given explicitly.

--reflog

Pretend as if all objects mentioned by reflogs are listed on the command line as `<commit>`.

--ignore-missing

Upon seeing an invalid object name in the input, pretend as if the bad input was not given.

--bisect

Pretend as if the bad bisection ref `refs/bisect/bad` was listed and as if it was followed by `--not` and the good bisection refs `refs/bisect/good-*` on the command line. Cannot be combined with --first-parent.

--stdin

In addition to the *<commit>* listed on the command line, read them from the standard input. If a -- separator is seen, stop reading commits and start reading paths to limit the result.

--cherry-mark

Like `--cherry-pick` (see below) but mark equivalent commits with `=` rather than omitting them, and inequivalent ones with `+`.

--cherry-pick

Omit any commit that introduces the same change as another commit on the “other side” when the set of commits are limited with symmetric difference.

For example, if you have two branches, `A` and `B`, a usual way to list all commits on only one side of them is with `--left-right` (see the example below in the description of the `--left-right` option). However, it shows the commits that were cherry-picked from the other branch (for example, “3rd on b” may be cherry-picked from branch A). With this option, such pairs of commits are excluded from the output.

--left-only

--right-only

List only commits on the respective side of a symmetric range, i.e. only those which would be marked `<` resp. `>` by `--left-right`.

For example, `--cherry-pick --right-only A...B` omits those commits from `B` which are in `A` or are patch-equivalent to a commit in `A`. In other words, this lists the `+` commits from `git cherry A B`. More precisely, `--cherry-pick --right-only --no-merges` gives the exact list.

--cherry

A synonym for `--right-only --cherry-mark --no-merges`; useful to limit the output to the commits on our side and mark those that have been applied to the other side of a forked history with `git log --cherry upstream...mybranch`, similar to `git cherry upstream mybranch`.

-g

--walk-reflogs

Instead of walking the commit ancestry chain, walk reflog entries from the most recent one to older ones. When this option is used you cannot specify commits to exclude (that is, *^commit*, *commit1..commit2*, and *commit1...commit2* notations cannot be used).

With `--pretty` format other than `oneline` (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. By default, *commit@{Nth}* notation is used in the output. When the starting commit is specified as *commit@{now}*, output also uses *commit@{timestamp}* notation instead. Under `--pretty=oneline`, the commit message is prefixed with this information on the same line. This option cannot be combined with `--reverse`. See also git-reflog[1].

--merge

After a failed merge, show refs that touch files having a conflict and don't exist on all heads to merge.

--boundary

Output excluded boundary commits. Boundary commits are prefixed with `-`.

History Simplification

Sometimes you are only interested in parts of the history, for example the commits modifying a particular <path>. But there are two parts of *History Simplification*, one part is selecting the commits and the other is how to do it, as there are various strategies to simplify the history.

The following options select the commits to be shown:

<paths>

Commits modifying the given <paths> are selected.

--simplify-by-decoration

Commits that are referred by some branch or tag are selected.

Note that extra commits can be shown to give a meaningful history.

The following options affect the way the simplification is performed:

Default mode

Simplifies the history to the simplest history explaining the final state of the tree. Simplest because it prunes some side branches if the end result is the same (i.e. merging branches with the same content)

--full-history

Same as the default mode, but does not prune some history.

--dense

Only the selected commits are shown, plus some to have a meaningful history.

--sparse

All commits in the simplified history are shown.

--simplify-merges

Additional option to `--full-history` to remove some needless merges from the resulting history, as there are no selected commits contributing to this merge.

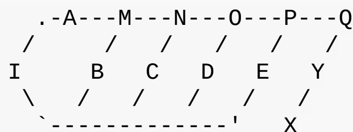
--ancestry-path

When given a range of commits to display (e.g. *commit1..commit2* or *commit2 ^commit1*), only display commits that exist directly on the ancestry chain between the *commit1* and *commit2*, i.e. commits that are both descendants of *commit1*, and ancestors of *commit2*.

A more detailed explanation follows.

Suppose you specified `foo` as the <paths>. We shall call commits that modify `foo` !TREESAME, and the rest TREESAME. (In a diff filtered for `foo`, they look different and equal, respectively.)

In the following, we will always refer to the same example history to illustrate the differences between simplification settings. We assume that you are filtering for a file `foo` in this commit graph:



The horizontal line of history A---Q is taken to be the first parent of each merge. The commits are:

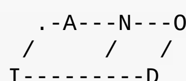
- `I` is the initial commit, in which `foo` exists with contents “asdf”, and a file `quux` exists with contents “quux”. Initial commits are compared to an empty tree, so `I` is !TREESAME.
- In `A`, `foo` contains just “foo”.
- `B` contains the same change as `A`. Its merge `M` is trivial and hence TREESAME to all parents.
- `C` does not change `foo`, but its merge `N` changes it to “foobar”, so it is not TREESAME to any parent.
- `D` sets `foo` to “baz”. Its merge `O` combines the strings from `N` and `D` to “foobarbaz”; i.e., it is not TREESAME to any parent.
- `E` changes `quux` to “xyzzy”, and its merge `P` combines the strings to “quux xyzzy”. `P` is TREESAME to `O`, but not to `E`.
- `X` is an independent root commit that added a new file `side`, and `Y` modified it. `Y` is TREESAME to `X`. Its merge `Q` added `side` to `P`, and `Q` is TREESAME to `P`, but not to `Y`.

`rev-list` walks backwards through history, including or excluding commits based on whether `--full-history` and/or parent rewriting (via `--parents` or `--children`) are used. The following settings are available.

Default mode

Commits are included if they are not TREESAME to any parent (though this can be changed, see `--sparse` below). If the commit was a merge, and it was TREESAME to one parent, follow only that parent. (Even if there are several TREESAME parents, follow only one of them.) Otherwise, follow all parents.

This results in:



Note how the rule to only follow the TREESAME parent, if one is available, removed `B` from consideration entirely. `C` was considered via `N`, but is TREESAME. Root commits are compared to an empty tree, so `I` is !TREESAME.

Parent/child relations are only visible with `--parents`, but that does not affect the commits selected in default mode, so we have shown the parent lines.

--full-history without parent rewriting

This mode differs from the default in one point: always follow all parents of a merge, even if it is TREESAME to one of them. Even if more than one side of the merge has commits that are included, this does not imply that the merge itself is! In the example, we get

```
I  A  B  N  D  O  P  Q
```

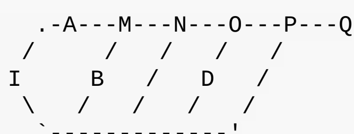
`M` was excluded because it is TREESAME to both parents. `E`, `C` and `B` were all walked, but only `B` was !TREESAME, so the others do not appear.

Note that without parent rewriting, it is not really possible to talk about the parent/child relationships between the commits, so we show them disconnected.

--full-history with parent rewriting

Ordinary commits are only included if they are !TREESAME (though this can be changed, see `--sparse` below).

Merges are always included. However, their parent list is rewritten: Along each parent, prune away commits that are not included themselves. This results in



```
  .-A---M---N---O---P---Q
 /     /   /   /   /
I     B   /   D   /
 \   /   /   /   /
  `-------------'
```

Compare to `--full-history` without rewriting above. Note that `E` was pruned away because it is TREESAME, but the parent list of P was rewritten to contain `E`'s parent `I`. The same happened for `C` and `N`, and `X`, `Y` and `Q`.

In addition to the above settings, you can change whether TREESAME affects inclusion:

--dense

Commits that are walked are included if they are not TREESAME to any parent.

--sparse

All commits that are walked are included.

Note that without `--full-history`, this still simplifies merges: if one of the parents is TREESAME, we follow only that one, so the other sides of the merge are never walked.

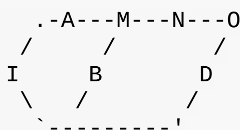
--simplify-merges

First, build a history graph in the same way that `--full-history` with parent rewriting does (see above).

Then simplify each commit `C` to its replacement `C'` in the final history according to the following rules:

- Set `C'` to `C`.
- Replace each parent `P` of `C'` with its simplification `P'`. In the process, drop parents that are ancestors of other parents or that are root commits TREESAME to an empty tree, and remove duplicates, but take care to never drop all parents that we are TREESAME to.
- If after this parent rewriting, `C'` is a root or merge commit (has zero or >1 parents), a boundary commit, or !TREESAME, it remains. Otherwise, it is replaced with its only parent.

The effect of this is best shown by way of comparing to `--full-history` with parent rewriting. The example turns into:



```
  .-A---M---N---O
 /     /       /
I     B       D
 \   /       /
  `---------'
```

Note the major differences in `N`, `P`, and `Q` over `--full-history`:

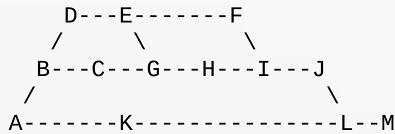
- `N`'s parent list had `I` removed, because it is an ancestor of the other parent `M`. Still, `N` remained because it is !TREESAME.
- `P`'s parent list similarly had `I` removed. `P` was then removed completely, because it had one parent and is TREESAME.
- `Q`'s parent list had `Y` simplified to `X`. `X` was then removed, because it was a TREESAME root. `Q` was then removed completely, because it had one parent and is TREESAME.

Finally, there is a fifth simplification mode available:

--ancestry-path

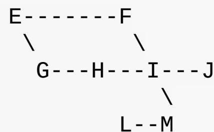
Limit the displayed commits to those directly on the ancestry chain between the “from” and “to” commits in the given commit range. I.e. only display commits that are ancestor of the “to” commit and descendants of the “from” commit.

As an example use case, consider the following commit history:


```
    D---E-------F
   /     \       \
  B---C---G---H---I---J
 /                     \
A-------K---------------L--M
```

A regular *D..M* computes the set of commits that are ancestors of `M`, but excludes the ones that are ancestors of `D`. This is useful to see what happened to the history leading to `M` since `D`, in the sense that “what does `M` have that did not exist in `D`”. The result in this example would be all the commits, except `A` and `B` (and `D` itself, of course).

When we want to find out what commits in `M` are contaminated with the bug introduced by `D` and need fixing, however, we might want to view only the subset of *D..M* that are actually descendants of `D`, i.e. excluding `C` and `K`. This is exactly what the `--ancestry-path` option does. Applied to the *D..M* range, it results in:


```
    E-------F
     \       \
      G---H---I---J
               \
              L--M
```

The `--simplify-by-decoration` option allows you to view only the big picture of the topology of the history, by omitting commits that are not referenced by tags. Commits are marked as !TREESAME (in other words, kept after history simplification rules described above) if (1) they are referenced by tags, or (2) they change the contents of the paths given on the command line. All other commits are marked as TREESAME (subject to be simplified away).

Commit Ordering

By default, the commits are shown in reverse chronological order.

--date-order

Show no parents before all of its children are shown, but otherwise show commits in the commit timestamp order.

--author-date-order

Show no parents before all of its children are shown, but otherwise show commits in the author timestamp order.

--topo-order

Show no parents before all of its children are shown, and avoid showing commits on multiple lines of history intermixed.

For example, in a commit history like this:

```
    ---1----2----4----7
    \            \
     3----5----6----8---
```

where the numbers denote the order of commit timestamps, `git rev-list` and friends with `--date-order` show the commits in the timestamp order: 8 7 6 5 4 3 2 1.

With `--topo-order`, they would show 8 6 5 3 7 4 2 1 (or 8 7 4 2 6 5 3 1); some older commits are shown before newer ones in order to avoid showing the commits from two parallel development track mixed together.

--reverse

Output the commits in reverse order. Cannot be combined with `--walk-reflogs`.

Object Traversal

These options are mostly targeted for packing of Git repositories.

--no-walk[=(sorted|unsorted)]

Only show the given commits, but do not traverse their ancestors. This has no effect if a range is specified. If the argument `unsorted` is given, the commits are shown in the order they were given on the command line. Otherwise (if `sorted` or no argument was given), the commits are shown in reverse chronological order by commit time. Cannot be combined with `--graph`.

--do-walk

Overrides a previous `--no-walk`.

Commit Formatting

pretty-options.txt

--relative-date

Synonym for `--date=relative`.

--date=<format>

Only takes effect for dates shown in human-readable format, such as when using `--pretty`. `log.date` config variable sets a default value for the log command's `--date` option. By default, dates are shown in the original time zone (either committer's or author's). If `-local` is appended to the format (e.g., `iso-local`), the user's local time zone is used instead.

`--date=relative` shows dates relative to the current time, e.g. “2 hours ago”. The `-local` option cannot be used with `--raw` or `--relative`.

`--date=local` is an alias for `--date=default-local`.

`--date=iso` (or `--date=iso8601`) shows timestamps in a ISO 8601-like format. The differences to the strict ISO 8601 format are:

- a space instead of the `T` date/time delimiter
- a space between time and time zone
- no colon between hours and minutes of the time zone

`--date=iso-strict` (or `--date=iso8601-strict`) shows timestamps in strict ISO 8601 format.

- `--date=rfc` (or `--date=rfc2822`) shows timestamps in RFC 2822 format, often found in email messages.
- `--date=short` shows only the date, but not the time, in `YYYY-MM-DD` format.
- `--date=raw` shows the date in the internal raw Git format `%s %z` format.
- `--date=format:...` feeds the format `...` to your system `strftime`. Use `--date=format:%c` to show the date in your system locale's preferred format. See the `strftime` manual for a complete list of format placeholders. When using `-local`, the correct syntax is `--date=format-local:...`.
- `--date=default` is the default format, and is similar to `--date=rfc2822`, with a few exceptions:
- there is no comma after the day-of-week
- the time zone is omitted when the local time zone is used

--parents

Print also the parents of the commit (in the form "commit parent…"). Also enables parent rewriting, see *History Simplification* below.

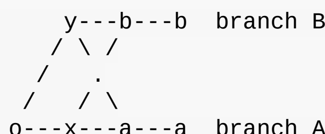
--children

Print also the children of the commit (in the form "commit child…"). Also enables parent rewriting, see *History Simplification* below.

--left-right

Mark which side of a symmetric diff a commit is reachable from. Commits from the left side are prefixed with `<` and those from the right with `>`. If combined with `--boundary`, those commits are prefixed with `-`.

For example, if you have this topology:



```
        y---b---b  branch B
       / \ /
      /   .
     /   / \
    o---x---a---a  branch A
```

you would get an output like this:

```
$ git rev-list --left-right --boundary --pretty=oneline A...B

&gt;bbbbbbb... 3rd on b
&gt;bbbbbbb... 2nd on b
&lt;aaaaaaa... 3rd on a
&lt;aaaaaaa... 2nd on a
-yyyyyyy... 1st on b
-xxxxxxx... 1st on a
```

--graph

Draw a text-based graphical representation of the commit history on the left hand side of the output. This may cause extra lines to be printed in between commits, in order for the graph history to be drawn properly. Cannot be combined with `--no-walk`.

This enables parent rewriting, see *History Simplification* below.

This implies the `--topo-order` option by default, but the `--date-order` option may also be specified.

--show-linear-break[=<barrier>]

When --graph is not used, all history branches are flattened which can make it hard to see that the two consecutive commits do not belong to a linear branch. This option puts a barrier in between them in that case. If `<barrier>` is specified, it is the string that will be shown instead of the default one.

Diff Formatting

Listed below are options that control the formatting of diff output. Some of them are specific to git-rev-list[1], however other diff options may be given. See git-diff-files[1] for more options.

-c

With this option, diff output for a merge commit shows the differences from each of the parents to the merge result simultaneously instead of showing pairwise diff between a parent and the result one at a time. Furthermore, it lists only files which were modified from all parents.

--cc

This flag implies the `-c` option and further compresses the patch output by omitting uninteresting hunks whose contents in the parents have only two variants and the merge result picks one of them without modification.

-m

This flag makes the merge commits show the full diff like regular commits; for each merge parent, a separate log entry and diff is generated. An exception is that only diff against the first parent is shown when `--first-parent` option is given; in that case, the output represents the changes the merge brought *into* the then-current branch.

-r

Show recursive diffs.

-t

Show the tree objects in the diff output. This implies `-r` .

PRETTY FORMATS

If the commit is a merge, and if the pretty-format is not *oneline*, *email* or *raw*, an additional line is inserted before the *Author:* line. This line begins with "Merge: " and the sha1s of ancestral commits are printed, separated by spaces. Note that the listed commits may not necessarily be the list of the **direct** parent commits if you have limited your view of history: for example, if you are only interested in changes related to a certain directory or file.

There are several built-in formats, and you can define additional formats by setting a pretty.<name> config option to either another format name, or a *format:* string, as described below (see git-config[1]). Here are the details of the built-in formats:

- *oneline*

  ```
  &lt;sha1&gt; &lt;title line&gt;
  ```

  This is designed to be as compact as possible.

- *short*

  ```
  commit &lt;sha1&gt;
  Author: &lt;author&gt;
  ```

  ```
  &lt;title line&gt;
  ```

- *medium*

  ```
  commit &lt;sha1&gt;
  Author: &lt;author&gt;
  Date:   &lt;author date&gt;
  ```

  ```
  &lt;title line&gt;
  ```

  ```
  &lt;full commit message&gt;
  ```

- *full*

  ```
  commit &lt;sha1&gt;
  Author: &lt;author&gt;
  Commit: &lt;committer&gt;
  ```

  ```
  &lt;title line&gt;
  ```

  ```
  &lt;full commit message&gt;
  ```

- *fuller*

  ```
  commit &lt;sha1&gt;
  Author:     &lt;author&gt;
  AuthorDate: &lt;author date&gt;
  Commit:     &lt;committer&gt;
  CommitDate: &lt;committer date&gt;
  ```

  ```
  &lt;title line&gt;
  ```

  ```
  &lt;full commit message&gt;
  ```

- *email*

```
From &lt;sha1&gt; &lt;date&gt;
From: &lt;author&gt;
Date: &lt;author date&gt;
Subject: [PATCH] &lt;title line&gt;
```

```
&lt;full commit message&gt;
```

- *raw*

  The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the SHA-1s are displayed in full, regardless of whether --abbrev or --no-abbrev are used, and *parents* information show the true parent commits, without taking grafts or history simplification into account. Note that this format affects the way commits are displayed, but not the way the diff is shown e.g. with `git log --raw`. To get full object names in a raw diff format, use `--no-abbrev`.

- *format:<string>*

  The *format:<string>* format allows you to specify which information you want to show. It works a little bit like printf format, with the notable exception that you get a newline with *%n* instead of *\n*.

  E.g, *format:"The author of %h was %an, %ar%nThe title was >>%s<<%n"* would show something like this:

  ```
  The author of fe6e0ee was Junio C Hamano, 23 hours ago
  The title was &gt;&gt;t4119: test autocomputing -p&lt;n&gt; for traditional diff inpu
  ```

  The placeholders are:

  - *%H*: commit hash
  - *%h*: abbreviated commit hash
  - *%T*: tree hash
  - *%t*: abbreviated tree hash
  - *%P*: parent hashes
  - *%p*: abbreviated parent hashes
  - *%an*: author name
  - *%aN*: author name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
  - *%ae*: author email

- *%aE*: author email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ad*: author date (format respects --date= option)
- *%aD*: author date, RFC2822 style
- *%ar*: author date, relative
- *%at*: author date, UNIX timestamp
- *%ai*: author date, ISO 8601-like format
- *%aI*: author date, strict ISO 8601 format
- *%cn*: committer name
- *%cN*: committer name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ce*: committer email
- *%cE*: committer email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%cd*: committer date (format respects --date= option)
- *%cD*: committer date, RFC2822 style
- *%cr*: committer date, relative
- *%ct*: committer date, UNIX timestamp
- *%ci*: committer date, ISO 8601-like format
- *%cI*: committer date, strict ISO 8601 format
- *%d*: ref names, like the --decorate option of git-log[1]
- *%D*: ref names without the " (", ")" wrapping.
- *%e*: encoding
- *%s*: subject
- *%f*: sanitized subject line, suitable for a filename
- *%b*: body
- *%B*: raw body (unwrapped subject and body)
- *%N*: commit notes
- *%GG*: raw verification message from GPG for a signed commit

- *%G?*: show "G" for a Good signature, "B" for a Bad signature, "U" for a good, untrusted signature and "N" for no signature
- *%GS*: show the name of the signer for a signed commit
- *%GK*: show the key used to sign a signed commit
- *%gD*: reflog selector, e.g., `refs/stash@{1}`
- *%gd*: shortened reflog selector, e.g., `stash@{1}`
- *%gn*: reflog identity name
- *%gN*: reflog identity name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ge*: reflog identity email
- *%gE*: reflog identity email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%gs*: reflog subject
- *%Cred*: switch color to red
- *%Cgreen*: switch color to green
- *%Cblue*: switch color to blue
- *%Creset*: reset color
- *%C(…)*: color specification, as described in color.branch.* config option; adding `auto,` at the beginning will emit color only when colors are enabled for log output (by `color.diff`, `color.ui`, or `--color`, and respecting the `auto` settings of the former if we are going to a terminal). `auto` alone (i.e. `%C(auto)`) will turn on auto coloring on the next placeholders until the color is switched again.
- *%m*: left, right or boundary mark
- *%n*: newline
- *%%*: a raw *%*
- *%x00*: print a byte from a hex code
- *%w([<w>[,<i1>[,<i2>]]])*: switch line wrapping, like the -w option of git-shortlog[1].
- *%<(<N>[,trunc|ltrunc|mtrunc])*: make the next placeholder take at least N columns, padding spaces on the right if necessary. Optionally truncate at the beginning (ltrunc), the middle (mtrunc) or the end (trunc) if the output is longer than N columns. Note that truncating only works correctly with N >= 2.

- *%<|(<N>)*: make the next placeholder take at least until Nth columns, padding spaces on the right if necessary
- *%>(<N>)*, *%>|(<N>)*: similar to *%<(<N>)*, *%<|(<N>)* respectively, but padding spaces on the left
- *%>>(<N>)*, *%>>|(<N>)*: similar to *%>(<N>)*, *%>|(<N>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces
- *%><(<N>)*, *%><|(<N>)*: similar to *% <(<N>)*, *%<|(<N>)* respectively, but padding both sides (i.e. the text is centered)

> Note
>
> Some placeholders may depend on other options given to the revision traversal engine. For example, the `%g*` reflog options will insert an empty string unless we are traversing reflog entries (e.g., by `git log -g`). The `%d` and `%D` placeholders will use the "short" decoration format if `--decorate` was not already provided on the command line.

If you add a `+` (plus sign) after *%* of a placeholder, a line-feed is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

If you add a `-` (minus sign) after *%* of a placeholder, line-feeds that immediately precede the expansion are deleted if and only if the placeholder expands to an empty string.

If you add a ` ` (space) after *%* of a placeholder, a space is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

- *tformat:*

  The *tformat:* format works exactly like *format:*, except that it provides "terminator" semantics instead of "separator" semantics. In other words, each commit has the message terminator character (usually a newline) appended, rather than a separator placed between entries. This means that the final entry of a single-line format will be properly terminated with a new line, just as the "oneline" format does. For example:

  ```
  $ git log -2 --pretty=format:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973 -- NO NEWLINE

  $ git log -2 --pretty=tformat:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973
  ```

In addition, any unrecognized string that has a `%` in it is interpreted as if it has `tformat:` in front of it. For example, these two are equivalent:

```
$ git log -2 --pretty=tformat:%h 4da45bef
$ git log -2 --pretty=%h 4da45bef
```

COMMON DIFF OPTIONS

-p

-u

--patch

Generate patch (see section on generating patches).

-s

--no-patch

Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw

For each commit, show a summary of changes using the raw diff format. See the "RAW OUTPUT FORMAT" section of git-diff[1]. This is different from showing the log itself in raw format, which you can achieve with `--format=raw`.

--patch-with-raw

Synonym for `-p --raw`.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

`default` , `myers`

The basic greedy diff algorithm. Currently, this is the default.

`minimal`

Spend extra time to make sure the smallest possible diff is produced.

`patience`

Use "patience diff" algorithm when generating patches.

`histogram`

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured diff.algorithm variable to a non-default value and want to use the default one, then you have to use `--diff-algorithm=default` option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by `<width>` . The width of the filename part can be limited by giving another width `<name-width>` after a comma. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (does not affect `git format-patch`). By giving a third parameter `<count>` , you can limit the output to the first `<count>` lines, followed by `...` if there are more.

These parameters can also be set individually with `--stat-width=<width>` , `--stat-name-width=<name-width>` and `--stat-count=<count>` .

--numstat

Similar to `--stat` , but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two `-` instead of saying `0 0` .

--shortstat

Output only the last line of the `--stat` format containing total number of modified files, as well as number of added and deleted lines.

--dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of `--dirstat` can be customized by passing it a comma separated list of parameters. The defaults are controlled by the `diff.dirstat` configuration variable (see git-config[1]). The following parameters are available:

`changes`

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

`lines`

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive `--dirstat` behavior than the `changes` behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other `--*stat` options.

`files`

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest `--dirstat` behavior, since it does not have to look at the file contents at all.

`cumulative`

Count changes in a child directory for the parent directory as well. Note that when using `cumulative`, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the `noncumulative` parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: `--dirstat=files,10,cumulative`.

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for `-p --stat`.

-z

Separate the commits with NULs instead of with new newlines.

Also, when `--raw` or `--numstat` has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only

Show only names of changed files.

--name-status

Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]

Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule[1] `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--color[=<when>]

Show colored diff. `--color` (i.e. without =<*when*>) is the same as `--color=always`. <*when*> can be one of `always`, `never`, or `auto`.

--no-color

Turn off colored diff. It is the same as `--color=never`.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see `--word-diff-regex` below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies `--color`.

plain

Show words as `[-removed-]` and `{+added+}`. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a `+` / `-` / `` ` character at the beginning of the line and extending to the end of the line. Newl``
~` on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies `--word-diff` unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append `|[^[:space:]]` to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, `--word-diff-regex=.` will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see gitattributes[1] or git-config[1]. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to `--word-diff=color` plus (if a regex was specified) `--word-diff-regex=<regex>`.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--check

Warn if changes introduce whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors on lines specified by <kind> in the color specified by `color.diff.whitespace`. <kind> is a comma separated list of `old`, `new`, `context`. When this option is not given, only whitespace errors in `new` lines are highlighted. E.g. `--ws-error-highlight=new,old` highlights whitespace errors on both deleted and added lines. `all` can be used as a short-hand for `old,new,context`.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to `--full-index`, output a binary diff that can be applied with `git-apply`.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. Non default number of digits can be specified with `--abbrev=<n>`.

-B[<n>][/<m>]

--break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything

new, and the number `m` controls this aspect of the -B option (defaults to 60%). `-B/70%` specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number `n` controls this aspect of the -B option (defaults to 50%). `-B20%` specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>]

--find-renames[=<n>]

If generating diffs, detect and report renames for each commit. For following files across renames while traversing history, see `--follow`. If `n` is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, `-M90%` means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a `%` sign, the number is to be read as a fraction, with a decimal point before it. I.e., `-M5` becomes 0.5, and is thus the same as `-M50%`. Similarly, `-M05` is the same as `-M5%`. To limit detection to exact renames, use `-M100%`. The default similarity index is 50%.

-C[<n>]

--find-copies[=<n>]

Detect copies as well as renames. See also `--find-copies-harder`. If `n` is specified, it has the same meaning as for `-M<n>`.

--find-copies-harder

For performance reasons, by default, `-C` option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one `-C` option has the same effect.

-D

--irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and `/dev/null`. The resulting patch is not meant to be applied with `patch` or `git apply`; this is solely for people who want to just concentrate on reviewing the text after

the change. In addition, the output obviously lack enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with `-B`, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The `-M` and `-C` options require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (`A`), Copied (`C`), Deleted (`D`), Modified (`M`), Renamed (`R`), have their type (i.e. regular file, symlink, submodule, …) changed (`T`), are Unmerged (`U`), are Unknown (`X`), or have had their pairing Broken (`B`). Any combination of the filter characters (including none) can be used. When `*` (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter's use.

It is useful when you're looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into `-S`, and keep going until you get the very first version of the block.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between `-S<regex> --pickaxe-regex` and `-G<regex>`, consider a commit with the following diff in the same file:

```
+    return !regexec(regexp, two-&gt;ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

See the *pickaxe* entry in gitdiffcore[7] for more information.

--pickaxe-all

When `-S` or `-G` finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to `-S` as an extended POSIX regular expression to match.

-O<orderfile>

Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config[1]). To cancel `diff.orderFile`, use `-O/dev/null`.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument.

-a

--text

Treat all files as text.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b

--ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w

--ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other.

-W

--function-context

Show whole surrounding functions of changes.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with gitattributes[5], you need to use this option with git-log[1] and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv

--no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See gitattributes[5] for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for git-diff[1] and git-log[1], but not for git-format-patch[1] or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in git-config[1] or gitmodules[5]. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

For more detailed explanation on these common options, see also gitdiffcore[7].

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The `a/` and `b/` filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, `/dev/null` is *not* used in place of the `a/` or `b/` filenames.

   When rename/copy is involved, `file1` and `file2` show the name of the source file of the rename/copy and the name of the file that rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

   ```
   old mode &lt;mode&gt;
   new mode &lt;mode&gt;
   deleted file mode &lt;mode&gt;
   new file mode &lt;mode&gt;
   copy from &lt;path&gt;
   copy to &lt;path&gt;
   rename from &lt;path&gt;
   rename to &lt;path&gt;
   similarity index &lt;number&gt;
   dissimilarity index &lt;number&gt;
   index &lt;hash&gt;..&lt;hash&gt; &lt;mode&gt;
   ```

   File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the `a/` and `b/` prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the SHA-1 checksum before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

4. All the `file1` files in the output refer to files before the commit, and all the `file2` files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

combined diff format

Any diff-generating command can take the `-c` or `--cc` option to produce a *combined diff* when showing a merge. This is the default format when showing merges with git-diff[1] or git-show[1]. Note also that you can give the `-m` option to any of these commands to force generation of diffs with individual parents of a merge.

A *combined diff* format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
    return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +    unsigned char sha1[20];
 +    struct commit *cmit;
    struct commit_list *list;
    static int initialized = 0;
    struct commit_name *n;

 +    if (get_sha1(arg, sha1) < 0)
 +        usage(describe_usage);
 +    cmit = lookup_commit_reference(sha1);
 +    if (!cmit)
 +        usage(describe_usage);
 +
    if (!initialized) {
        initialized = 1;
        for_each_ref(get_name);
```

1. It is preceded with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index &lt;hash&gt;;,&lt;hash&gt;;..&lt;hash&gt;;
   mode &lt;mode&gt;;,&lt;mode&gt;;..&lt;mode&gt;;
   new file mode &lt;mode&gt;;
   deleted file mode &lt;mode&gt;;,&lt;mode&gt;;
   ```

   The `mode <mode>;,<mode>;..<mode>;` line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.

4. Chunk header format is modified to prevent people from accidentally feeding it to `patch -p1`. Combined diff format was created for review of merge commit changes, and was not meant for apply. The change is similar to the change in the extended *index* header:

```
@@@ &lt;from-file-range&gt; &lt;from-file-range&gt; &lt;to-file-range&gt; @@@
```

There are (number of parents + 1) `@` characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has `-` (minus — appears in A but removed in B), `+` (plus — missing in A but added to B), or `" "` (space — unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A `-` character in the column N means that the line appears in fileN but it does not appear in the result. A `+` character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two `-` removals from both file1 and file2, plus `++` to mean one line that was added does not appear in either file1 or file2). Also eight other lines are the same from file1 but do not appear in file2 (hence prefixed with `+`).

When shown by `git diff-tree -c`, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by `git diff-files -c`, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

EXAMPLES

`git log --no-merges`

Show the whole commit history, but skip any merges

`git log v2.6.12.. include/scsi drivers/scsi`

Show all commits since version *v2.6.12* that changed any file in the `include/scsi` or `drivers/scsi` subdirectories

```
git log --since="2 weeks ago" -- gitk
```

Show the changes during the last two weeks to the file *gitk*. The “--” is necessary to avoid confusion with the **branch** named *gitk*

```
git log --name-status release..test
```

Show the commits that are in the "test" branch but not yet in the "release" branch, along with the list of paths each commit modifies.

```
git log --follow builtin/rev-list.c
```

Shows the commits that changed `builtin/rev-list.c`, including those commits that occurred before the file was given its present name.

```
git log --branches --not --remotes=origin
```

Shows all commits that are in any of local branches but not in any of remote-tracking branches for *origin* (what you have that origin doesn’t).

```
git log master --not --remotes=*/master
```

Shows all commits that are in local master but not in any remote repository master branches.

```
git log -p -m --first-parent
```

Shows the history including change diffs, but only from the “main branch” perspective, skipping commits that come from merged branches, and showing full diffs of changes introduced by the merges. This makes sense only when following a strict policy of merging all topic branches when staying on a single integration branch.

```
git log -L '/int main/',/^}/:main.c
```

Shows how the function `main()` in the file `main.c` evolved over time.

```
git log -3
```

Limits the number of commits to show to 3.

DISCUSSION

Git is to some extent character encoding agnostic.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (`.git/config` (see git-config[1]), gitignore[5],

gitattributes[5] and gitmodules[5]).

Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that use legacy extended ASCII encodings. However, repositories created on such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.

- Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but *not* UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

Although we encourage that the commit log messages are encoded in UTF-8, both the core and Git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, Git does not forbid it. However, there are a few things to keep in mind.

1. *git commit* and *git commit-tree* issues a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have i18n.commitencoding in `.git/config` file, like this:

   ```
   [i18n]
       commitencoding = ISO-8859-1
   ```

   Commit objects created with the above setting record the value of `i18n.commitencoding` in its `encoding` header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.

2. *git log*, *git show*, *git blame* and friends look at the `encoding` header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with `i18n.logoutputencoding` in `.git/config` file, like this:

   ```
   [i18n]
       logoutputencoding = ISO-8859-1
   ```

   If you do not have this configuration variable, the value of `i18n.commitencoding` is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

CONFIGURATION

See git-config[1] for core variables and git-diff[1] for settings related to diff generation.

format.pretty

Default for the `--format` option. (See *Pretty Formats* above.) Defaults to `medium`.

i18n.logOutputEncoding

Encoding to use when displaying logs. (See *Discussion* above.) Defaults to the value of `i18n.commitEncoding` if set, and UTF-8 otherwise.

log.date

Default format for human-readable dates. (Compare the `--date` option.) Defaults to "default", which means to write dates like `Sat May 8 19:35:34 2010 -0500`.

log.follow

If `true`, `git log` will act as if the `--follow` option was used when a single <path> is given. This has the same limitations as `--follow`, i.e. it cannot be used to follow multiple files and does not work well on non-linear history.

log.showRoot

If `false`, `git log` and related commands will not treat the initial commit as a big creation event. Any root commits in `git log -p` output would be shown without a diff attached. The default is `true`.

mailmap.*

See git-shortlog[1].

notes.displayRef

Which refs, in addition to the default set by `core.notesRef` or *GIT_NOTES_REF*, to read notes from when showing commit messages with the `log` family of commands. See git-notes[1].

May be an unabbreviated ref name or a glob and may be specified multiple times. A warning will be issued for refs that do not exist, but a glob that does not match any refs is silently ignored.

This setting can be disabled by the `--no-notes` option, overridden by the *GIT_NOTES_DISPLAY_REF* environment variable, and overridden by the `--notes=<ref>` option.

GIT

Part of the git[1] suite

stash

NAME

git-stash - Stash the changes in a dirty working directory away

SYNOPSIS

```
git stash list [<options>]
git stash show [<stash>]
git stash drop [-q|--quiet] [<stash>]
git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]
git stash branch <branchname> [<stash>]
git stash [save [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet]
         [-u|--include-untracked] [-a|--all] [<message>]]
git stash clear
git stash create [<message>]
git stash store [-m|--message <message>] [-q|--quiet] <commit>
```

DESCRIPTION

Use `git stash` when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the `HEAD` commit.

The modifications stashed away by this command can be listed with `git stash list`, inspected with `git stash show`, and restored (potentially on top of a different commit) with `git stash apply`. Calling `git stash` without any arguments is equivalent to `git stash save`. A stash is by default listed as "WIP on *branchname* …", but you can give a more descriptive message on the command line when you create one.

The latest stash you created is stored in `refs/stash`; older stashes are found in the reflog of this reference and can be named using the usual reflog syntax (e.g. `stash@{0}` is the most recently created stash, `stash@{1}` is the one before it, `stash@{2.hours.ago}` is also possible).

OPTIONS

save [-p|--patch] [-k|--[no-]keep-index] [-u|--include-untracked] [-a|--all] [-q|--quiet] [<message>]

Save your local modifications to a new *stash*, and run `git reset --hard` to revert them. The <message> part is optional and gives the description along with the stashed state. For quickly making a snapshot, you can omit *both* "save" and <message>, but giving only <message> does not trigger this action to prevent a misspelled subcommand from making an unwanted stash.

If the `--keep-index` option is used, all changes already added to the index are left intact.

If the `--include-untracked` option is used, all untracked files are also stashed and then cleaned up with `git clean`, leaving the working directory in a very clean state. If the `--all` option is used instead then the ignored files are stashed and cleaned in addition to the untracked files.

With `--patch`, you can interactively select hunks from the diff between HEAD and the working tree to be stashed. The stash entry is constructed such that its index state is the same as the index state of your repository, and its worktree contains only the changes you selected interactively. The selected changes are then rolled back from your worktree. See the “Interactive Mode” section of git-add[1] to learn how to operate the `--patch` mode.

The `--patch` option implies `--keep-index`. You can use `--no-keep-index` to override this.

list [<options>]

List the stashes that you currently have. Each *stash* is listed with its name (e.g. `stash@{0}` is the latest stash, `stash@{1}` is the one before, etc.), the name of the branch that was current when the stash was made, and a short description of the commit the stash was based on.

```
stash@{0}: WIP on submit: 6ebd0e2... Update git-stash documentation
stash@{1}: On master: 9cc0589... Add git-stash
```

The command takes options applicable to the *git log* command to control what is shown and how. See git-log[1].

show [<stash>]

Show the changes recorded in the stash as a diff between the stashed state and its original parent. When no `<stash>` is given, shows the latest one. By default, the command shows the diffstat, but it will accept any format known to *git diff* (e.g., `git stash show -p stash@{1}` to view the second most recent stash in patch form). You can use stash.showStat and/or stash.showPatch config variables to change the default behavior.

pop [--index] [-q|--quiet] [<stash>]

Remove a single stashed state from the stash list and apply it on top of the current working tree state, i.e., do the inverse operation of `git stash save`. The working directory must match the index.

Applying the state can fail with conflicts; in this case, it is not removed from the stash list. You need to resolve the conflicts by hand and call `git stash drop` manually afterwards.

If the `--index` option is used, then tries to reinstate not only the working tree's changes, but also the index's ones. However, this can fail, when you have conflicts (which are stored in the index, where you therefore can no longer apply the changes as they were originally).

When no `<stash>` is given, `stash@{0}` is assumed, otherwise `<stash>` must be a reference of the form `stash@{<revision>}`.

apply [--index] [-q|--quiet] [<stash>]

Like `pop`, but do not remove the state from the stash list. Unlike `pop`, `<stash>` may be any commit that looks like a commit created by `stash save` or `stash create`.

branch <branchname> [<stash>]

Creates and checks out a new branch named `<branchname>` starting from the commit at which the `<stash>` was originally created, applies the changes recorded in `<stash>` to the new working tree and index. If that succeeds, and `<stash>` is a reference of the form `stash@{<revision>}`, it then drops the `<stash>`. When no `<stash>` is given, applies the latest one.

This is useful if the branch on which you ran `git stash save` has changed enough that `git stash apply` fails due to conflicts. Since the stash is applied on top of the commit that was HEAD at the time `git stash` was run, it restores the originally stashed state with no conflicts.

clear

Remove all the stashed states. Note that those states will then be subject to pruning, and may be impossible to recover (see *Examples* below for a possible strategy).

drop [-q|--quiet] [<stash>]

Remove a single stashed state from the stash list. When no `<stash>` is given, it removes the latest one. i.e. `stash@{0}`, otherwise `<stash>` must be a valid stash log reference of the form `stash@{<revision>}`.

create

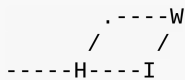
Create a stash (which is a regular commit object) and return its object name, without storing it anywhere in the ref namespace. This is intended to be useful for scripts. It is probably not the command you want to use; see "save" above.

store

Store a given stash created via *git stash create* (which is a dangling merge commit) in the stash ref, updating the stash reflog. This is intended to be useful for scripts. It is probably not the command you want to use; see "save" above.

DISCUSSION

A stash is represented as a commit whose tree records the state of the working directory, and its first parent is the commit at `HEAD` when the stash was created. The tree of the second parent records the state of the index when the stash is made, and it is made a child of the `HEAD` commit. The ancestry graph looks like this:



```
       .----W
      /    /
-----H----I
```

where `H` is the `HEAD` commit, `I` is a commit that records the state of the index, and `W` is a commit that records the state of the working tree.

EXAMPLES

Pulling into a dirty tree

When you are in the middle of something, you learn that there are upstream changes that are possibly relevant to what you are doing. When your local changes do not conflict with the changes in the upstream, a simple `git pull` will let you move forward.

However, there are cases in which your local changes do conflict with the upstream changes, and `git pull` refuses to overwrite your changes. In such a case, you can stash your changes away, perform a pull, and then unstash, like this:

```
$ git pull
 ...
file foobar not up to date, cannot merge.
$ git stash
$ git pull
$ git stash pop
```

Interrupted workflow

When you are in the middle of something, your boss comes in and demands that you fix something immediately. Traditionally, you would make a commit to a temporary branch to store your changes away, and return to your original branch to make the emergency fix, like this:

```
# ... hack hack hack ...
$ git checkout -b my_wip
$ git commit -a -m "WIP"
$ git checkout master
$ edit emergency fix
$ git commit -a -m "Fix in a hurry"
$ git checkout my_wip
$ git reset --soft HEAD^
# ... continue hacking ...
```

You can use *git stash* to simplify the above, like this:

```
# ... hack hack hack ...
$ git stash
$ edit emergency fix
$ git commit -a -m "Fix in a hurry"
$ git stash pop
# ... continue hacking ...
```

Testing partial commits

You can use `git stash save --keep-index` when you want to make two or more commits out of the changes in the work tree, and you want to test each change before committing:

```
# ... hack hack hack ...
$ git add --patch foo            # add just first part to the index
$ git stash save --keep-index    # save all other changes to the stash
$ edit/build/test first part
$ git commit -m 'First part'     # commit fully tested change
$ git stash pop                  # prepare to work on all other changes
# ... repeat above five steps until one commit remains ...
$ edit/build/test remaining parts
$ git commit foo -m 'Remaining parts'
```

Recovering stashes that were cleared/dropped erroneously

If you mistakenly drop or clear stashes, they cannot be recovered through the normal safety mechanisms. However, you can try the following incantation to get a list of stashes that are still in your repository, but not reachable any more:

```
git fsck --unreachable |
grep commit | cut -d\  -f3 |
xargs git log --merges --no-walk --grep=WIP
```

SEE ALSO

git-checkout[1], git-commit[1], git-reflog[1], git-reset[1]

GIT

Part of the git[1] suite

tag

NAME

git-tag - Create, list, delete or verify a tag object signed with GPG

SYNOPSIS

```
git tag [-a | -s | -u <keyid>] [-f] [-m <msg> | -F <file>]
    <tagname> [<commit> | <object>]
git tag -d <tagname>…
git tag [-n[<num>]] -l [--contains <commit>] [--points-at <object>]
    [--column[=<options>] | --no-column] [--create-reflog] [--sort=<key>]
    [--format=<format>] [--[no-]merged [<commit>]] [<pattern>…]
git tag -v <tagname>…
```

DESCRIPTION

Add a tag reference in `refs/tags/` , unless `-d/-l/-v` is given to delete, list or verify tags.

Unless `-f` is given, the named tag must not yet exist.

If one of `-a` , `-s` , or `-u <keyid>` is passed, the command creates a *tag* object, and requires a tag message. Unless `-m <msg>` or `-F <file>` is given, an editor is started for the user to type in the tag message.

If `-m <msg>` or `-F <file>` is given and `-a` , `-s` , and `-u <keyid>` are absent, `-a` is implied.

Otherwise just a tag reference for the SHA-1 object name of the commit object is created (i.e. a lightweight tag).

A GnuPG signed tag object will be created when `-s` or `-u <keyid>` is used. When `-u <keyid>` is not used, the committer identity for the current user is used to find the GnuPG key for signing. The configuration variable `gpg.program` is used to specify custom GnuPG binary.

Tag objects (created with `-a` , `-s` , or `-u`) are called "annotated" tags; they contain a creation date, the tagger name and e-mail, a tagging message, and an optional GnuPG signature. Whereas a "lightweight" tag is simply a name for an object (usually a commit object).

Annotated tags are meant for release while lightweight tags are meant for private or temporary object labels. For this reason, some git commands for naming objects (like `git describe`) will ignore lightweight tags by default.

OPTIONS

-a

--annotate

Make an unsigned, annotated tag object

-s

--sign

Make a GPG-signed tag, using the default e-mail address's key.

-u <keyid>

--local-user=<keyid>

Make a GPG-signed tag, using the given key.

-f

--force

Replace an existing tag with the given name (instead of failing)

-d

--delete

Delete existing tags with the given names.

-v

--verify

Verify the gpg signature of the given tag names.

-n<num>

<num> specifies how many lines from the annotation, if any, are printed when using -l. The default is not to print any annotation lines. If no number is given to `-n`, only the first line is printed. If the tag is not annotated, the commit message is displayed instead.

-l <pattern>

--list <pattern>

List tags with names that match the given pattern (or all if no pattern is given). Running "git tag" without arguments also lists all tags. The pattern is a shell wildcard (i.e., matched using fnmatch(3)). Multiple patterns may be given; if any of them matches, the tag is shown.

--sort=<key>

Sort based on the key given. Prefix `-` to sort in descending order of the value. You may use the --sort=<key> option multiple times, in which case the last key becomes the primary key. Also supports "version:refname" or "v:refname" (tag names are treated as versions). The "version:refname" sort order can also be affected by the "versionsort.prereleaseSuffix" configuration variable. The keys supported are the same as those in `git for-each-ref`. Sort order defaults to the value configured for the *tag.sort* variable if it exists, or lexicographic order otherwise. See git-config[1].

--column[=<options>]

--no-column

Display tag listing in columns. See configuration variable column.tag for option syntax. `--column` and `--no-column` without options are equivalent to *always* and *never* respectively.

This option is only applicable when listing tags without annotation lines.

--contains [<commit>]

Only list tags which contain the specified commit (HEAD if not specified).

--points-at <object>

Only list tags of the given object.

-m <msg>

--message=<msg>

Use the given tag message (instead of prompting). If multiple `-m` options are given, their values are concatenated as separate paragraphs. Implies `-a` if none of `-a`, `-s`, or `-u <keyid>` is given.

-F <file>

--file=<file>

Take the tag message from the given file. Use - to read the message from the standard input. Implies `-a` if none of `-a`, `-s`, or `-u <keyid>` is given.

--cleanup=<mode>

This option sets how the tag message is cleaned up. The *<mode>* can be one of *verbatim*, *whitespace* and *strip*. The *strip* mode is default. The *verbatim* mode does not change message at all, *whitespace* removes just leading/trailing whitespace lines and *strip* removes both whitespace and commentary.

--create-reflog

Create a reflog for the tag.

<tagname>

The name of the tag to create, delete, or describe. The new tag name must pass all checks defined by git-check-ref-format[1]. Some of these checks may restrict the characters allowed in a tag name.

<commit>

<object>

The object that the new tag will refer to, usually a commit. Defaults to HEAD.

<format>

A string that interpolates `%(fieldname)` from the object pointed at by a ref being shown. The format is the same as that of git-for-each-ref[1]. When unspecified, defaults to `%(refname:strip=2)`.

--[no-]merged [<commit>]

Only list tags whose tips are reachable, or not reachable if *--no-merged* is used, from the specified commit (*HEAD* if not specified).

CONFIGURATION

By default, *git tag* in sign-with-default mode (-s) will use your committer identity (of the form `Your Name <your@email.address>`) to find a key. If you want to use a different default key, you can specify it in the repository configuration as follows:

```
[user]
    signingKey = <gpg-keyid>
```

DISCUSSION

On Re-tagging

What should you do when you tag a wrong commit and you would want to re-tag?

If you never pushed anything out, just re-tag it. Use "-f" to replace the old one. And you're done.

But if you have pushed things out (or others could just read your repository directly), then others will have already seen the old tag. In that case you can do one of two things:

1. The sane thing. Just admit you screwed up, and use a different name. Others have already seen one tag-name, and if you keep the same name, you may be in the situation that two people both have "version X", but they actually have *different* "X"'s. So just call it "X.1" and be done with it.

2. The insane thing. You really want to call the new version "X" too, *even though* others have already seen the old one. So just use *git tag -f* again, as if you hadn't already published the old one.

However, Git does **not** (and it should not) change tags behind users back. So if somebody already got the old tag, doing a *git pull* on your tree shouldn't just make them overwrite the old one.

If somebody got a release tag from you, you cannot just change the tag for them by updating your own one. This is a big security issue, in that people MUST be able to trust their tag-names. If you really want to do the insane thing, you need to just fess up to it, and tell people that you messed up. You can do that by making a very public announcement saying:

```
Ok, I messed up, and I pushed out an earlier version tagged as X. I
then fixed something, and retagged the *fixed* tree as X again.

If you got the wrong tag, and want the new one, please delete
the old one and fetch the new one by doing:

    git tag -d X
    git fetch origin tag X

to get my updated tag.

You can test which tag you have by doing

    git rev-parse X

which should return 0123456789abcdef.. if you have the new version.

Sorry for the inconvenience.
```

Does this seem a bit complicated? It **should** be. There is no way that it would be correct to just "fix" it automatically. People need to know that their tags might have been changed.

On Automatic following

If you are following somebody else's tree, you are most likely using remote-tracking branches (`refs/heads/origin` in traditional layout, or `refs/remotes/origin/master` in the separate-remote layout). You usually want the tags from the other end.

On the other hand, if you are fetching because you would want a one-shot merge from somebody else, you typically do not want to get tags from there. This happens more often for people near the toplevel but not limited to them. Mere mortals when pulling from each other do not necessarily want to automatically get private anchor point tags from the other person.

Often, "please pull" messages on the mailing list just provide two pieces of information: a repo URL and a branch name; this is designed to be easily cut&pasted at the end of a *git fetch* command line:

```
Linus, please pull from

    git://git..../proj.git master

to get the following updates...
```

becomes:

```
$ git pull git://git..../proj.git master
```

In such a case, you do not want to automatically follow the other person's tags.

One important aspect of Git is its distributed nature, which largely means there is no inherent "upstream" or "downstream" in the system. On the face of it, the above example might seem to indicate that the tag namespace is owned by the upper echelon of people and that tags only flow downwards, but that is not the case. It only shows that the usage pattern determines who are interested in whose tags.

A one-shot pull is a sign that a commit history is now crossing the boundary between one circle of people (e.g. "people who are primarily interested in the networking part of the kernel") who may have their own set of tags (e.g. "this is the third release candidate from the networking group to be proposed for general consumption with 2.6.21 release") to another circle of people (e.g. "people who integrate various subsystem improvements"). The latter are usually not interested in the detailed tags used internally in the former group (that is what "internal" means). That is why it is desirable not to follow tags automatically in this case.

It may well be that among networking people, they may want to exchange the tags internal to their group, but in that workflow they are most likely tracking each other's progress by having remote-tracking branches. Again, the heuristic to automatically follow such tags is a good thing.

On Backdating Tags

If you have imported some changes from another VCS and would like to add tags for major releases of your work, it is useful to be able to specify the date to embed inside of the tag object; such data in the tag object affects, for example, the ordering of tags in the gitweb interface.

To set the date used in future tag objects, set the environment variable GIT_COMMITTER_DATE (see the later discussion of possible values; the most common form is "YYYY-MM-DD HH:MM").

For example:

```
$ GIT_COMMITTER_DATE="2006-10-02 10:31" git tag -s v1.0.1
```

DATE FORMATS

The GIT_AUTHOR_DATE, GIT_COMMITTER_DATE environment variables support the following date formats:

Git internal format

It is `<unix timestamp> <time zone offset>`, where `<unix timestamp>` is the number of seconds since the UNIX epoch. `<time zone offset>` is a positive or negative offset from UTC. For example CET (which is 2 hours ahead UTC) is `+0200`.

RFC 2822

The standard email format as described by RFC 2822, for example `Thu, 07 Apr 2005 22:13:13 +0200`.

ISO 8601

Time and date specified by the ISO 8601 standard, for example `2005-04-07T22:13:13`. The parser accepts a space instead of the `T` character as well.

> Note
>
> In addition, the date part is accepted in the following formats: `YYYY.MM.DD`, `MM/DD/YYYY` and `DD.MM.YYYY`.

SEE ALSO

git-check-ref-format[1]. git-config[1].

GIT

Part of the git[1] suite

Sharing and Updating Projects

fetch

NAME

git-fetch - Download objects and refs from another repository

SYNOPSIS

```
git fetch [<options>] [<repository> [<refspec>…]]
git fetch [<options>] <group>
git fetch --multiple [<options>] [(<repository> | <group>)…]
git fetch --all [<options>]
```

DESCRIPTION

Fetch branches and/or tags (collectively, "refs") from one or more other repositories, along with the objects necessary to complete their histories. Remote-tracking branches are updated (see the description of <refspec> below for ways to control this behavior).

By default, any tag that points into the histories being fetched is also fetched; the effect is to fetch tags that point at branches that you are interested in. This default behavior can be changed by using the --tags or --no-tags options or by configuring remote.<name>.tagOpt. By using a refspec that fetches tags explicitly, you can fetch tags that do not point into branches you are interested in as well.

git fetch can fetch from either a single named repository or URL, or from several repositories at once if <group> is given and there is a remotes.<group> entry in the configuration file. (See git-config[1]).

When no remote is specified, by default the `origin` remote will be used, unless there’s an upstream branch configured for the current branch.

The names of refs that are fetched, together with the object names they point at, are written to `.git/FETCH_HEAD`. This information may be used by scripts or other git commands, such as git-pull[1].

OPTIONS

--all

Fetch all remotes.

-a

--append

Append ref names and object names of fetched refs to the existing contents of `.git/FETCH_HEAD`. Without this option old data in `.git/FETCH_HEAD` will be overwritten.

--depth=<depth>

Limit fetching to the specified number of commits from the tip of each remote branch history. If fetching to a *shallow* repository created by `git clone` with `--depth=<depth>` option (see git-clone[1]), deepen or shorten the history to the specified number of commits. Tags for the deepened commits are not fetched.

--unshallow

If the source repository is complete, convert a shallow repository to a complete one, removing all the limitations imposed by shallow repositories.

If the source repository is shallow, fetch as much as possible so that the current repository has the same history as the source repository.

--update-shallow

By default when fetching from a shallow repository, `git fetch` refuses refs that require updating .git/shallow. This option updates .git/shallow and accept such refs.

--dry-run

Show what would be done, without making any changes.

-f

--force

When *git fetch* is used with `<rbranch>:<lbranch>` refspec, it refuses to update the local branch `<lbranch>` unless the remote branch `<rbranch>` it fetches is a descendant of `<lbranch>`. This option overrides that check.

-k

--keep

Keep downloaded pack.

--multiple

Allow several <repository> and <group> arguments to be specified. No <refspec>s may be specified.

-p

--prune

After fetching, remove any remote-tracking references that no longer exist on the remote. Tags are not subject to pruning if they are fetched only because of the default tag auto-following or due to a --tags option. However, if tags are fetched due to an explicit refspec (either on the command line or in the remote configuration, for example if the remote was cloned with the --mirror option), then they are also subject to pruning.

-n

--no-tags

By default, tags that point at objects that are downloaded from the remote repository are fetched and stored locally. This option disables this automatic tag following. The default behavior for a remote may be specified with the remote.<name>.tagOpt setting. See git-config[1].

--refmap=<refspec>

When fetching refs listed on the command line, use the specified refspec (can be given more than once) to map the refs to remote-tracking branches, instead of the values of `remote.*.fetch` configuration variables for the remote repository. See section on "Configured Remote-tracking Branches" for details.

-t

--tags

Fetch all tags from the remote (i.e., fetch remote tags `refs/tags/*` into local tags with the same name), in addition to whatever else would otherwise be fetched. Using this option alone does not subject tags to pruning, even if --prune is used (though tags may be pruned anyway if they are also the destination of an explicit refspec; see *--prune*).

--recurse-submodules[=yes|on-demand|no]

This option controls if and under what conditions new commits of populated submodules should be fetched too. It can be used as a boolean option to completely disable recursion when set to *no* or to unconditionally recurse into all populated submodules when set to *yes*, which is the default when this option is used without any value. Use *on-demand* to only recurse into a populated submodule when the superproject retrieves a commit that updates the submodule’s reference to a commit that isn’t already in the local submodule clone.

-j

--jobs=<n>

Number of parallel children to be used for fetching submodules. Each will fetch from different submodules, such that fetching many submodules will be faster. By default submodules will be fetched one at a time.

--no-recurse-submodules

Disable recursive fetching of submodules (this has the same effect as using the *--recurse-submodules=no* option).

--submodule-prefix=<path>

Prepend <path> to paths printed in informative messages such as "Fetching submodule foo". This option is used internally when recursing over submodules.

--recurse-submodules-default=[yes|on-demand]

This option is used internally to temporarily provide a non-negative default value for the --recurse-submodules option. All other methods of configuring fetch’s submodule recursion (such as settings in gitmodules[5] and git-config[1]) override this option, as does specifying --[no-]recurse-submodules directly.

-u

--update-head-ok

By default *git fetch* refuses to update the head which corresponds to the current branch. This flag disables the check. This is purely for the internal use for *git pull* to communicate with *git fetch*, and unless you are implementing your own Porcelain you are not supposed to use it.

--upload-pack <upload-pack>

When given, and the repository to fetch from is handled by *git fetch-pack*, *--exec=<upload-pack>* is passed to the command to specify non-default path for the command run on the other end.

-q

--quiet

Pass --quiet to git-fetch-pack and silence any other internally used git commands. Progress is not reported to the standard error stream.

-v

--verbose

Be verbose.

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

-4

--ipv4

Use IPv4 addresses only, ignoring IPv6 addresses.

-6

--ipv6

Use IPv6 addresses only, ignoring IPv4 addresses.

<repository>

The "remote" repository that is the source of a fetch or pull operation. This parameter can be either a URL (see the section GIT URLS below) or the name of a remote (see the section REMOTES below).

<group>

A name referring to a list of repositories as the value of remotes.<group> in the configuration file. (See git-config[1]).

<refspec>

Specifies which refs to fetch and which local refs to update. When no <refspec>s appear on the command line, the refs to fetch are read from `remote.<repository>.fetch` variables instead (see CONFIGURED REMOTE-TRACKING BRANCHES below).

The format of a <refspec> parameter is an optional plus `+`, followed by the source ref <src>, followed by a colon `:`, followed by the destination ref <dst>. The colon can be omitted when <dst> is empty.

`tag <tag>` means the same as `refs/tags/<tag>:refs/tags/<tag>`; it requests fetching everything up to the given tag.

The remote ref that matches <src> is fetched, and if <dst> is not empty string, the local ref that matches it is fast-forwarded using <src>. If the optional plus `+` is used, the local ref is updated even if it does not result in a fast-forward update.

> Note
>
> When the remote branch you want to fetch is known to be rewound and rebased regularly, it is expected that its new tip will not be descendant of its previous tip (as stored in your remote-tracking branch the last time you fetched). You would want to use the `+` sign to indicate non-fast-forward updates will be needed for such branches. There is no way to determine or declare that a branch will be made available in a repository with this behavior; the pulling user simply must know this is the expected usage pattern for a branch.

urls.txt

REMOTES

The name of one of the following can be used instead of a URL as `<repository>` argument:

- a remote in the Git configuration file: `$GIT_DIR/config` ,
- a file in the `$GIT_DIR/remotes` directory, or
- a file in the `$GIT_DIR/branches` directory.

All of these also allow you to omit the refspec from the command line because they each contain a refspec which git will use by default.

Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using git-remote[1], git-config[1] or even by a manual edit to the `$GIT_DIR/config` file. The URL of this remote will be used to access the repository. The refspec of this remote will be used by default when you do not provide a refspec on the command line. The entry in the config file would appear like this:

```
	[remote "<name>"]
		url = <url>
		pushurl = <pushurl>
		push = <refspec>
		fetch = <refspec>
```

The `<pushurl>` is used for pushes only. It is optional and defaults to `<url>` .

Named file in `$GIT_DIR/remotes`

You can choose to provide the name of a file in $GIT_DIR/remotes. The URL in this file will be used to access the repository. The refspec in this file will be used as default when you do not provide a refspec on the command line. This file should have the following format:

```
URL: one of the above URL format
Push: <refspec>
Pull: <refspec>
```

Push: lines are used by *git push* and Pull: lines are used by *git pull* and *git fetch*. Multiple Push: and Pull: lines may be specified for additional branch mappings.

Named file in $GIT_DIR/branches

You can choose to provide the name of a file in $GIT_DIR/branches. The URL in this file will be used to access the repository. This file should have the following format:

```
<url>#<head>
```

<url> is required; #<head> is optional.

Depending on the operation, git will use one of the following refspecs, if you don’t provide one on the command line. <branch> is the name of this file in $GIT_DIR/branches and <head> defaults to master.

git fetch uses:

```
refs/heads/<head>:refs/heads/<branch>
```

git push uses:

```
HEAD:refs/heads/<head>
```

CONFIGURED REMOTE-TRACKING BRANCHES

You often interact with the same remote repository by regularly and repeatedly fetching from it. In order to keep track of the progress of such a remote repository, git fetch allows you to configure remote.<repository>.fetch configuration variables.

Typically such a variable may look like this:

```
[remote "origin"]
    fetch = +refs/heads/*:refs/remotes/origin/*
```

This configuration is used in two ways:

- When `git fetch` is run without specifying what branches and/or tags to fetch on the command line, e.g. `git fetch origin` or `git fetch`, `remote.<repository>.fetch` values are used as the refspecs—they specify which refs to fetch and which local refs to update. The example above will fetch all branches that exist in the `origin` (i.e. any ref that matches the left-hand side of the value, `refs/heads/*`) and update the corresponding remote-tracking branches in the `refs/remotes/origin/*` hierarchy.
- When `git fetch` is run with explicit branches and/or tags to fetch on the command line, e.g. `git fetch origin master`, the <refspec>s given on the command line determine what are to be fetched (e.g. `master` in the example, which is a short-hand for `master:`, which in turn means "fetch the *master* branch but I do not explicitly say what remote-tracking branch to update with it from the command line"), and the example command will fetch *only* the *master* branch. The `remote.<repository>.fetch` values determine which remote-tracking branch, if any, is updated. When used in this way, the `remote.<repository>.fetch` values do not have any effect in deciding *what* gets fetched (i.e. the values are not used as refspecs when the command-line lists refspecs); they are only used to decide *where* the refs that are fetched are stored by acting as a mapping.

The latter use of the `remote.<repository>.fetch` values can be overridden by giving the `--refmap=<refspec>` parameter(s) on the command line.

EXAMPLES

- Update the remote-tracking branches:

  ```
  $ git fetch origin
  ```

  The above command copies all branches from the remote refs/heads/ namespace and stores them to the local refs/remotes/origin/ namespace, unless the branch. <name>.fetch option is used to specify a non-default refspec.

- Using refspecs explicitly:

  ```
  $ git fetch origin +pu:pu maint:tmp
  ```

  This updates (or creates, as necessary) branches `pu` and `tmp` in the local repository by fetching from the branches (respectively) `pu` and `maint` from the remote repository.

The `pu` branch will be updated even if it is does not fast-forward, because it is prefixed with a plus sign; `tmp` will not be.

- Peek at a remote's branch, without configuring the remote in your local repository:

```
$ git fetch git://git.kernel.org/pub/scm/git/git.git maint
$ git log FETCH_HEAD
```

The first command fetches the `maint` branch from the repository at `git://git.kernel.org/pub/scm/git/git.git` and the second command uses `FETCH_HEAD` to examine the branch with git-log[1]. The fetched objects will eventually be removed by git's built-in housekeeping (see git-gc[1]).

BUGS

Using --recurse-submodules can only fetch new commits in already checked out submodules right now. When e.g. upstream added a new submodule in the just fetched commits of the superproject the submodule itself can not be fetched, making it impossible to check out that submodule later without having to do a fetch again. This is expected to be fixed in a future Git version.

SEE ALSO

git-pull[1]

GIT

Part of the git[1] suite

pull

NAME

git-pull - Fetch from and integrate with another repository or a local branch

SYNOPSIS

```
git pull [options] [<repository> [<refspec>…]]
```

DESCRIPTION

Incorporates changes from a remote repository into the current branch. In its default mode, `git pull` is shorthand for `git fetch` followed by `git merge FETCH_HEAD` .

More precisely, *git pull* runs *git fetch* with the given parameters and calls *git merge* to merge the retrieved branch heads into the current branch. With `--rebase` , it runs *git rebase* instead of *git merge*.

<repository> should be the name of a remote repository as passed to git-fetch[1]. <refspec> can name an arbitrary remote ref (for example, the name of a tag) or even a collection of refs with corresponding remote-tracking branches (e.g., refs/heads/*:refs/remotes/origin/*), but usually it is the name of a branch in the remote repository.

Default values for <repository> and <branch> are read from the "remote" and "merge" configuration for the current branch as set by git-branch[1] `--track` .

Assume the following history exists and the current branch is " `master` ":

```
          A---B---C master on origin
         /
    D---E---F---G master
    ^
    origin/master in your repository
```

Then " `git pull` " will fetch and replay the changes from the remote `master` branch since it diverged from the local `master` (i.e., `E`) until its current commit (`C`) on top of `master` and record the result in a new commit along with the names of the two parent commits and a log message from the user describing the changes.

```
      A---B---C origin/master
     /         \
    D---E---F---G---H master
```

See git-merge[1] for details, including how conflicts are presented and handled.

In Git 1.7.0 or later, to cancel a conflicting merge, use `git reset --merge`. **Warning**: In older versions of Git, running *git pull* with uncommitted changes is discouraged: while possible, it leaves you in a state that may be hard to back out of in the case of a conflict.

If any of the remote changes overlap with local uncommitted changes, the merge will be automatically cancelled and the work tree untouched. It is generally best to get any local changes in working order before pulling or stash them away with git-stash[1].

OPTIONS

-q

--quiet

This is passed to both underlying git-fetch to squelch reporting of during transfer, and underlying git-merge to squelch output during merging.

-v

--verbose

Pass --verbose to git-fetch and git-merge.

--[no-]recurse-submodules[=yes|on-demand|no]

This option controls if new commits of all populated submodules should be fetched too (see git-config[1] and gitmodules[5]). That might be necessary to get the data needed for merging submodule commits, a feature Git learned in 1.7.3. Notice that the result of a merge will not be checked out in the submodule, "git submodule update" has to be called afterwards to bring the work tree up to date with the merge result.

Options related to merging

--commit

--no-commit

Perform the merge and commit the result. This option can be used to override --no-commit.

With --no-commit perform the merge but pretend the merge failed and do not autocommit, to give the user a chance to inspect and further tweak the merge result before committing.

--edit

-e

--no-edit

Invoke an editor before committing successful mechanical merge to further edit the auto-generated merge message, so that the user can explain and justify the merge. The `--no-edit` option can be used to accept the auto-generated message (this is generally discouraged).

Older scripts may depend on the historical behaviour of not allowing the user to edit the merge log message. They will see an editor opened when they run `git merge`. To make it easier to adjust such scripts to the updated behaviour, the environment variable `GIT_MERGE_AUTOEDIT` can be set to `no` at the beginning of them.

--ff

When the merge resolves as a fast-forward, only update the branch pointer, without creating a merge commit. This is the default behavior.

--no-ff

Create a merge commit even when the merge resolves as a fast-forward. This is the default behaviour when merging an annotated (and possibly signed) tag.

--ff-only

Refuse to merge and exit with a non-zero status unless the current `HEAD` is already up-to-date or the merge can be resolved as a fast-forward.

--log[=<n>]

--no-log

In addition to branch names, populate the log message with one-line descriptions from at most <n> actual commits that are being merged. See also git-fmt-merge-msg[1].

With --no-log do not list one-line descriptions from the actual commits being merged.

--stat

-n

--no-stat

Show a diffstat at the end of the merge. The diffstat is also controlled by the configuration option merge.stat.

With -n or --no-stat do not show a diffstat at the end of the merge.

--squash

--no-squash

Produce the working tree and index state as if a real merge happened (except for the merge information), but do not actually make a commit, move the `HEAD`, or record `$GIT_DIR/MERGE_HEAD` (to cause the next `git commit` command to create a merge commit). This allows you to create a single commit on top of the current branch whose effect is the same as merging another branch (or more in case of an octopus).

With --no-squash perform the merge and commit the result. This option can be used to override --squash.

-s <strategy>

--strategy=<strategy>

Use the given merge strategy; can be supplied more than once to specify them in the order they should be tried. If there is no `-s` option, a built-in list of strategies is used instead (*git merge-recursive* when merging a single head, *git merge-octopus* otherwise).

-X <option>

--strategy-option=<option>

Pass merge strategy specific option through to the merge strategy.

--verify-signatures

--no-verify-signatures

Verify that the commits being merged have good and trusted GPG signatures and abort the merge in case they do not.

--summary

--no-summary

Synonyms to --stat and --no-stat; these are deprecated and will be removed in the future.

-r

--rebase[=false|true|preserve|interactive]

When true, rebase the current branch on top of the upstream branch after fetching. If there is a remote-tracking branch corresponding to the upstream branch and the upstream branch was rebased since last fetched, the rebase uses that information to avoid rebasing non-local changes.

When set to preserve, rebase with the `--preserve-merges` option passed to `git rebase` so that locally created merge commits will not be flattened.

When false, merge the current branch into the upstream branch.

When `interactive`, enable the interactive mode of rebase.

See `pull.rebase`, `branch.<name>.rebase` and `branch.autoSetupRebase` in git-config[1] if you want to make `git pull` always use `--rebase` instead of merging.

> Note
>
> This is a potentially *dangerous* mode of operation. It rewrites history, which does not bode well when you published that history already. Do **not** use this option unless you have read git-rebase[1] carefully.

--no-rebase

Override earlier --rebase.

Options related to fetching

--all

Fetch all remotes.

-a

--append

Append ref names and object names of fetched refs to the existing contents of `.git/FETCH_HEAD`. Without this option old data in `.git/FETCH_HEAD` will be overwritten.

--depth=<depth>

Limit fetching to the specified number of commits from the tip of each remote branch history. If fetching to a *shallow* repository created by `git clone` with `--depth=<depth>` option (see git-clone[1]), deepen or shorten the history to the specified number of commits. Tags for the deepened commits are not fetched.

--unshallow

If the source repository is complete, convert a shallow repository to a complete one, removing all the limitations imposed by shallow repositories.

If the source repository is shallow, fetch as much as possible so that the current repository has the same history as the source repository.

--update-shallow

By default when fetching from a shallow repository, `git fetch` refuses refs that require updating .git/shallow. This option updates .git/shallow and accept such refs.

-f

--force

When *git fetch* is used with `<rbranch>:<lbranch>` refspec, it refuses to update the local branch `<lbranch>` unless the remote branch `<rbranch>` it fetches is a descendant of `<lbranch>`. This option overrides that check.

-k

--keep

Keep downloaded pack.

--no-tags

By default, tags that point at objects that are downloaded from the remote repository are fetched and stored locally. This option disables this automatic tag following. The default behavior for a remote may be specified with the remote.<name>.tagOpt setting. See git-config[1].

-u

--update-head-ok

By default *git fetch* refuses to update the head which corresponds to the current branch. This flag disables the check. This is purely for the internal use for *git pull* to communicate with *git fetch*, and unless you are implementing your own Porcelain you are not supposed to use it.

--upload-pack <upload-pack>

When given, and the repository to fetch from is handled by *git fetch-pack*, *--exec=<upload-pack>* is passed to the command to specify non-default path for the command run on the other end.

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

-4

--ipv4

Use IPv4 addresses only, ignoring IPv6 addresses.

-6

--ipv6

Use IPv6 addresses only, ignoring IPv4 addresses.

<repository>

The "remote" repository that is the source of a fetch or pull operation. This parameter can be either a URL (see the section GIT URLS below) or the name of a remote (see the section REMOTES below).

<refspec>

Specifies which refs to fetch and which local refs to update. When no <refspec>s appear on the command line, the refs to fetch are read from `remote.<repository>.fetch` variables instead (see git-fetch[1]).

The format of a <refspec> parameter is an optional plus `+`, followed by the source ref <src>, followed by a colon `:`, followed by the destination ref <dst>. The colon can be omitted when <dst> is empty.

`tag <tag>` means the same as `refs/tags/<tag>:refs/tags/<tag>`; it requests fetching everything up to the given tag.

The remote ref that matches <src> is fetched, and if <dst> is not empty string, the local ref that matches it is fast-forwarded using <src>. If the optional plus `+` is used, the local ref is updated even if it does not result in a fast-forward update.

> Note
>
> When the remote branch you want to fetch is known to be rewound and rebased regularly, it is expected that its new tip will not be descendant of its previous tip (as stored in your remote-tracking branch the last time you fetched). You would want to use the `+` sign to indicate non-fast-forward updates will be needed for such branches. There is no way to determine or declare that a branch will be made available in a repository with this behavior; the pulling user simply must know this is the expected usage pattern for a branch.
>
> Note
>
> There is a difference between listing multiple <refspec> directly on *git pull* command line and having multiple `remote.<repository>.fetch` entries in your configuration for a <repository> and running a *git pull* command without any explicit <refspec> parameters. <refspec>s listed explicitly on the command line are always merged into the current branch after fetching. In other words, if you list more than one remote ref, *git pull* will create an Octopus merge. On the other hand, if you do not list any explicit <refspec> parameter on the command line, *git pull* will fetch all the <refspec>s it finds in the `remote.<repository>.fetch` configuration and merge only the first <refspec> found into the current branch. This is because making an Octopus from remote refs is rarely done, while keeping track of multiple remote heads in one-go by fetching more than one is often useful.

urls.txt

REMOTES

The name of one of the following can be used instead of a URL as `<repository>` argument:

- a remote in the Git configuration file: `$GIT_DIR/config` ,
- a file in the `$GIT_DIR/remotes` directory, or
- a file in the `$GIT_DIR/branches` directory.

All of these also allow you to omit the refspec from the command line because they each contain a refspec which git will use by default.

Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using git-remote[1], git-config[1] or even by a manual edit to the `$GIT_DIR/config` file. The URL of this remote will be used to access the repository. The refspec of this remote will be used by default when you do not provide a refspec on the command line. The entry in the config file would appear like this:

```
[remote "<name>"]
    url = <url>
    pushurl = <pushurl>
    push = <refspec>
    fetch = <refspec>
```

The `<pushurl>` is used for pushes only. It is optional and defaults to `<url>`.

Named file in `$GIT_DIR/remotes`

You can choose to provide the name of a file in `$GIT_DIR/remotes`. The URL in this file will be used to access the repository. The refspec in this file will be used as default when you do not provide a refspec on the command line. This file should have the following format:

```
URL: one of the above URL format
Push: <refspec>
Pull: <refspec>
```

`Push:` lines are used by *git push* and `Pull:` lines are used by *git pull* and *git fetch*. Multiple `Push:` and `Pull:` lines may be specified for additional branch mappings.

Named file in `$GIT_DIR/branches`

You can choose to provide the name of a file in `$GIT_DIR/branches`. The URL in this file will be used to access the repository. This file should have the following format:

```
<url>#<head>
```

`<url>` is required; `#<head>` is optional.

Depending on the operation, git will use one of the following refspecs, if you don't provide one on the command line. `<branch>` is the name of this file in `$GIT_DIR/branches` and `<head>` defaults to `master`.

git fetch uses:

```
refs/heads/<head>:refs/heads/<branch>
```

git push uses:

```
HEAD:refs/heads/<head>
```

MERGE STRATEGIES

The merge mechanism (`git merge` and `git pull` commands) allows the backend *merge strategies* to be chosen with `-s` option. Some strategies can also take their own options, which can be passed by giving `-X<option>` arguments to `git merge` and/or `git pull`.

resolve

This can only resolve two heads (i.e. the current branch and another branch you pulled from) using a 3-way merge algorithm. It tries to carefully detect criss-cross merge ambiguities and is considered generally safe and fast.

recursive

This can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge. This has been reported to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from Linux 2.6 kernel development history. Additionally this can detect and handle merges involving renames. This is the default merge strategy when pulling or merging one branch.

The *recursive* strategy can take the following options:

ours

This option forces conflicting hunks to be auto-resolved cleanly by favoring *our* version. Changes from the other tree that do not conflict with our side are reflected to the merge result. For a binary file, the entire contents are taken from our side.

This should not be confused with the *ours* merge strategy, which does not even look at what the other tree contains at all. It discards everything the other tree did, declaring *our* history contains all that happened in it.

theirs

This is the opposite of *ours*.

patience

With this option, *merge-recursive* spends a little extra time to avoid mismerges that sometimes occur due to unimportant matching lines (e.g., braces from distinct functions). Use this when the branches to be merged have diverged wildly. See also git-diff[1] `--patience`.

diff-algorithm=[patience|minimal|histogram|myers]

Tells *merge-recursive* to use a different diff algorithm, which can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also git-diff[1] `--diff-algorithm`.

ignore-space-change

ignore-all-space

ignore-space-at-eol

Treats lines with the indicated type of whitespace change as unchanged for the sake of a three-way merge. Whitespace changes mixed with other changes to a line are not ignored. See also git-diff[1] `-b`, `-w`, and `--ignore-space-at-eol`.

- If *their* version only introduces whitespace changes to a line, *our* version is used;
- If *our* version introduces whitespace changes but *their* version includes a substantial change, *their* version is used;
- Otherwise, the merge proceeds in the usual way.

renormalize

This runs a virtual check-out and check-in of all three stages of a file when resolving a three-way merge. This option is meant to be used when merging branches with different clean filters or end-of-line normalization rules. See "Merging branches with differing checkin/checkout attributes" in gitattributes[5] for details.

no-renormalize

Disables the `renormalize` option. This overrides the `merge.renormalize` configuration variable.

no-renames

Turn off rename detection. See also git-diff[1] `--no-renames`.

find-renames[=<n>]

Turn on rename detection, optionally setting the similarity threshold. This is the default. See also git-diff[1] `--find-renames`.

rename-threshold=<n>

Deprecated synonym for `find-renames=<n>`.

subtree[=<path>]

This option is a more advanced form of *subtree* strategy, where the strategy makes a guess on how two trees must be shifted to match with each other when merging. Instead, the specified path is prefixed (or stripped from the beginning) to make the shape of two trees to match.

octopus

This resolves cases with more than two heads, but refuses to do a complex merge that needs manual resolution. It is primarily meant to be used for bundling topic branch heads together. This is the default merge strategy when pulling or merging more than one branch.

ours

This resolves any number of heads, but the resulting tree of the merge is always that of the current branch head, effectively ignoring all changes from all other branches. It is meant to be used to supersede old development history of side branches. Note that this is different from the -Xours option to the *recursive* merge strategy.

subtree

This is a modified recursive strategy. When merging trees A and B, if B corresponds to a subtree of A, B is first adjusted to match the tree structure of A, instead of reading the trees at the same level. This adjustment is also done to the common ancestor tree.

With the strategies that use 3-way merge (including the default, *recursive*), if a change is made on both branches, but later reverted on one of the branches, that change will be present in the merged result; some people find this behavior confusing. It occurs because only the heads and the merge base are considered when performing a merge, not the individual commits. The merge algorithm therefore considers the reverted change as no change at all, and substitutes the changed version instead.

DEFAULT BEHAVIOUR

Often people use `git pull` without giving any parameter. Traditionally, this has been equivalent to saying `git pull origin`. However, when configuration `branch.<name>.remote` is present while on branch `<name>`, that value is used instead of `origin`.

In order to determine what URL to use to fetch from, the value of the configuration `remote.<origin>.url` is consulted and if there is not any such variable, the value on `URL:` line in `$GIT_DIR/remotes/<origin>` file is used.

In order to determine what remote branches to fetch (and optionally store in the remote-tracking branches) when the command is run without any refspec parameters on the command line, values of the configuration variable `remote.<origin>.fetch` are consulted, and if there aren't any, `$GIT_DIR/remotes/<origin>` file is consulted and its `Pull:` lines are used. In addition to the refspec formats described in the OPTIONS section, you can have a globbing refspec that looks like this:

```
refs/heads/*:refs/remotes/origin/*
```

A globbing refspec must have a non-empty RHS (i.e. must store what were fetched in remote-tracking branches), and its LHS and RHS must end with `/*`. The above specifies that all remote branches are tracked using remote-tracking branches in `refs/remotes/origin/` hierarchy under the same name.

The rule to determine which remote branch to merge after fetching is a bit involved, in order not to break backward compatibility.

If explicit refspecs were given on the command line of `git pull`, they are all merged.

When no refspec was given on the command line, then `git pull` uses the refspec from the configuration or `$GIT_DIR/remotes/<origin>`. In such cases, the following rules apply:

1. If `branch.<name>.merge` configuration for the current branch `<name>` exists, that is the name of the branch at the remote site that is merged.
2. If the refspec is a globbing one, nothing is merged.
3. Otherwise the remote branch of the first refspec is merged.

EXAMPLES

- Update the remote-tracking branches for the repository you cloned from, then merge one of them into your current branch:

  ```
  $ git pull, git pull origin
  ```

  Normally the branch merged in is the HEAD of the remote repository, but the choice is determined by the branch.<name>.remote and branch.<name>.merge options; see git-config[1] for details.

- Merge into the current branch the remote branch `next`:

  ```
  $ git pull origin next
  ```

  This leaves a copy of `next` temporarily in FETCH_HEAD, but does not update any remote-tracking branches. Using remote-tracking branches, the same can be done by invoking fetch and merge:

  ```
  $ git fetch origin
  $ git merge origin/next
  ```

If you tried a pull which resulted in complex conflicts and would want to start over, you can recover with *git reset*.

BUGS

Using --recurse-submodules can only fetch new commits in already checked out submodules right now. When e.g. upstream added a new submodule in the just fetched commits of the superproject the submodule itself can not be fetched, making it impossible to check out that submodule later without having to do a fetch again. This is expected to be fixed in a future Git version.

SEE ALSO

git-fetch[1], git-merge[1], git-config[1]

GIT

Part of the git[1] suite

push

NAME

git-push - Update remote refs along with associated objects

SYNOPSIS

```
git push [--all | --mirror | --tags] [--follow-tags] [--atomic] [-n | --dry-run] [--recei
         [--repo=<repository>] [-f | --force] [-d | --delete] [--prune] [-v | --verbose]
         [-u | --set-upstream]
         [--[no-]signed|--sign=(true|false|if-asked)]
         [--force-with-lease[=<refname>[:<expect>]]]
         [--no-verify] [<repository> [<refspec>…]]
```

DESCRIPTION

Updates remote refs using local refs, while sending objects necessary to complete the given refs.

You can make interesting things happen to a repository every time you push into it, by setting up *hooks* there. See documentation for git-receive-pack[1].

When the command line does not specify where to push with the `<repository>` argument, `branch.*.remote` configuration for the current branch is consulted to determine where to push. If the configuration is missing, it defaults to *origin*.

When the command line does not specify what to push with `<refspec>...` arguments or `--all`, `--mirror`, `--tags` options, the command finds the default `<refspec>` by consulting `remote.*.push` configuration, and if it is not found, honors `push.default` configuration to decide what to push (See git-config[1] for the meaning of `push.default`).

When neither the command-line nor the configuration specify what to push, the default behavior is used, which corresponds to the `simple` value for `push.default`: the current branch is pushed to the corresponding upstream branch, but as a safety measure, the push is aborted if the upstream branch does not have the same name as the local one.

OPTIONS

<repository>

The "remote" repository that is destination of a push operation. This parameter can be either a URL (see the section GIT URLS below) or the name of a remote (see the section REMOTES below).

<refspec>…

Specify what destination ref to update with what source object. The format of a <refspec> parameter is an optional plus `+`, followed by the source object <src>, followed by a colon `:`, followed by the destination ref <dst>.

The <src> is often the name of the branch you would want to push, but it can be any arbitrary "SHA-1 expression", such as `master~4` or `HEAD` (see gitrevisions[7]).

The <dst> tells which ref on the remote side is updated with this push. Arbitrary expressions cannot be used here, an actual ref must be named. If `git push [<repository>]` without any `<refspec>` argument is set to update some ref at the destination with `<src>` with `remote.<repository>.push` configuration variable, `:<dst>` part can be omitted—such a push will update a ref that `<src>` normally updates without any `<refspec>` on the command line. Otherwise, missing `:<dst>` means to update the same ref as the `<src>`.

The object referenced by <src> is used to update the <dst> reference on the remote side. By default this is only allowed if <dst> is not a tag (annotated or lightweight), and then only if it can fast-forward <dst>. By having the optional leading `+`, you can tell Git to update the <dst> ref even if it is not allowed by default (e.g., it is not a fast-forward.) This does **not** attempt to merge <src> into <dst>. See EXAMPLES below for details.

`tag <tag>` means the same as `refs/tags/<tag>:refs/tags/<tag>`.

Pushing an empty <src> allows you to delete the <dst> ref from the remote repository.

The special refspec `:` (or `+:` to allow non-fast-forward updates) directs Git to push "matching" branches: for every branch that exists on the local side, the remote side is updated if a branch of the same name already exists on the remote side.

--all

Push all branches (i.e. refs under `refs/heads/`); cannot be used with other <refspec>.

--prune

Remove remote branches that don't have a local counterpart. For example a remote branch `tmp` will be removed if a local branch with the same name doesn't exist any more. This also respects refspecs, e.g. `git push --prune remote refs/heads/*:refs/tmp/*` would make sure that remote `refs/tmp/foo` will be removed if `refs/heads/foo` doesn't exist.

--mirror

Instead of naming each ref to push, specifies that all refs under `refs/` (which includes but is not limited to `refs/heads/`, `refs/remotes/`, and `refs/tags/`) be mirrored to the remote repository. Newly created local refs will be pushed to the remote end, locally updated refs will be force updated on the remote end, and deleted refs will be removed from the remote end. This is the default if the configuration option `remote.<remote>.mirror` is set.

-n

--dry-run

Do everything except actually send the updates.

--porcelain

Produce machine-readable output. The output status line for each ref will be tab-separated and sent to stdout instead of stderr. The full symbolic names of the refs will be given.

--delete

All listed refs are deleted from the remote repository. This is the same as prefixing all refs with a colon.

--tags

All refs under `refs/tags` are pushed, in addition to refspecs explicitly listed on the command line.

--follow-tags

Push all the refs that would be pushed without this option, and also push annotated tags in `refs/tags` that are missing from the remote but are pointing at commit-ish that are reachable from the refs being pushed. This can also be specified with configuration variable *push.followTags*. For more information, see *push.followTags* in git-config[1].

--[no-]signed

--sign=(true|false|if-asked)

GPG-sign the push request to update refs on the receiving side, to allow it to be checked by the hooks and/or be logged. If `false` or `--no-signed`, no signing will be attempted. If `true` or `--signed`, the push will fail if the server does not support signed pushes. If set to `if-asked`, sign if and only if the server supports signed pushes. The push will also fail if the actual call to `gpg --sign` fails. See git-receive-pack[1] for the details on the receiving end.

--[no-]atomic

Use an atomic transaction on the remote side if available. Either all refs are updated, or on error, no refs are updated. If the server does not support atomic pushes the push will fail.

--receive-pack=<git-receive-pack>

--exec=<git-receive-pack>

Path to the *git-receive-pack* program on the remote end. Sometimes useful when pushing to a remote repository over ssh, and you do not have the program in a directory on the default $PATH.

--[no-]force-with-lease

--force-with-lease=<refname>

--force-with-lease=<refname>:<expect>

Usually, "git push" refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it.

This option overrides this restriction if the current value of the remote ref is the expected value. "git push" fails otherwise.

Imagine that you have to rebase what you have already published. You will have to bypass the "must fast-forward" rule in order to replace the history you originally published with the rebased history. If somebody else built on top of your original history while you are rebasing, the tip of the branch at the remote may advance with her commit, and blindly pushing with `--force` will lose her work.

This option allows you to say that you expect the history you are updating is what you rebased and want to replace. If the remote ref still points at the commit you specified, you can be sure that no other people did anything to the ref. It is like taking a "lease" on the ref without explicitly locking it, and the remote ref is updated only if the "lease" is still valid.

`--force-with-lease` alone, without specifying the details, will protect all remote refs that are going to be updated by requiring their current value to be the same as the remote-tracking branch we have for them.

`--force-with-lease=<refname>`, without specifying the expected value, will protect the named ref (alone), if it is going to be updated, by requiring its current value to be the same as the remote-tracking branch we have for it.

`--force-with-lease=<refname>:<expect>` will protect the named ref (alone), if it is going to be updated, by requiring its current value to be the same as the specified value <expect> (which is allowed to be different from the remote-tracking branch we have for the

refname, or we do not even have to have such a remote-tracking branch when this form is used).

Note that all forms other than `--force-with-lease=<refname>:<expect>` that specifies the expected current value of the ref explicitly are still experimental and their semantics may change as we gain experience with this feature.

"--no-force-with-lease" will cancel all the previous --force-with-lease on the command line.

-f

--force

Usually, the command refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it. Also, when `--force-with-lease` option is used, the command refuses to update a remote ref whose current value does not match what is expected.

This flag disables these checks, and can cause the remote repository to lose commits; use it with care.

Note that `--force` applies to all the refs that are pushed, hence using it with `push.default` set to `matching` or with multiple push destinations configured with `remote.*.push` may overwrite refs other than the current branch (including local refs that are strictly behind their remote counterpart). To force a push to only one branch, use a `+` in front of the refspec to push (e.g `git push origin +master` to force a push to the `master` branch). See the `<refspec>...` section above for details.

--repo=<repository>

This option is equivalent to the <repository> argument. If both are specified, the command-line argument takes precedence.

-u

--set-upstream

For every branch that is up to date or successfully pushed, add upstream (tracking) reference, used by argument-less git-pull[1] and other commands. For more information, see *branch.<name>.merge* in git-config[1].

--[no-]thin

These options are passed to git-send-pack[1]. A thin transfer significantly reduces the amount of sent data when the sender and receiver share many of the same objects in common. The default is --thin.

-q

--quiet

Suppress all output, including the listing of updated refs, unless an error occurs. Progress is not reported to the standard error stream.

-v

--verbose

Run verbosely.

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

--no-recurse-submodules

--recurse-submodules=check|on-demand|no

May be used to make sure all submodule commits used by the revisions to be pushed are available on a remote-tracking branch. If *check* is used Git will verify that all submodule commits that changed in the revisions to be pushed are available on at least one remote of the submodule. If any commits are missing the push will be aborted and exit with non-zero status. If *on-demand* is used all submodules that changed in the revisions to be pushed will be pushed. If on-demand was not able to push all necessary revisions it will also be aborted and exit with non-zero status. A value of *no* or using *--no-recurse-submodules* can be used to override the push.recurseSubmodules configuration variable when no submodule recursion is required.

--[no-]verify

Toggle the pre-push hook (see githooks[5]). The default is --verify, giving the hook a chance to prevent the push. With --no-verify, the hook is bypassed completely.

-4

--ipv4

Use IPv4 addresses only, ignoring IPv6 addresses.

-6

--ipv6

Use IPv6 addresses only, ignoring IPv4 addresses.

urls.txt

REMOTES

The name of one of the following can be used instead of a URL as `<repository>` argument:

- a remote in the Git configuration file: `$GIT_DIR/config`,
- a file in the `$GIT_DIR/remotes` directory, or
- a file in the `$GIT_DIR/branches` directory.

All of these also allow you to omit the refspec from the command line because they each contain a refspec which git will use by default.

Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using git-remote[1], git-config[1] or even by a manual edit to the `$GIT_DIR/config` file. The URL of this remote will be used to access the repository. The refspec of this remote will be used by default when you do not provide a refspec on the command line. The entry in the config file would appear like this:

```
[remote "<name>"]
    url = <url>
    pushurl = <pushurl>
    push = <refspec>
    fetch = <refspec>
```

The `<pushurl>` is used for pushes only. It is optional and defaults to `<url>`.

Named file in `$GIT_DIR/remotes`

You can choose to provide the name of a file in `$GIT_DIR/remotes`. The URL in this file will be used to access the repository. The refspec in this file will be used as default when you do not provide a refspec on the command line. This file should have the following format:

```
URL: one of the above URL format
Push: <refspec>
Pull: <refspec>
```

`Push:` lines are used by *git push* and `Pull:` lines are used by *git pull* and *git fetch*. Multiple `Push:` and `Pull:` lines may be specified for additional branch mappings.

Named file in `$GIT_DIR/branches`

You can choose to provide the name of a file in `$GIT_DIR/branches`. The URL in this file will be used to access the repository. This file should have the following format:

```
    <url>#<head>
```

`<url>` is required; `#<head>` is optional.

Depending on the operation, git will use one of the following refspecs, if you don’t provide one on the command line. `<branch>` is the name of this file in `$GIT_DIR/branches` and `<head>` defaults to `master`.

git fetch uses:

```
    refs/heads/<head>:refs/heads/<branch>
```

git push uses:

```
    HEAD:refs/heads/<head>
```

OUTPUT

The output of "git push" depends on the transport method used; this section describes the output when pushing over the Git protocol (either locally or via ssh).

The status of the push is output in tabular form, with each line representing the status of a single ref. Each line is of the form:

```
 <flag> <summary> <from> -> <to> (<reason>)
```

If --porcelain is used, then each line of the output is of the form:

```
 <flag> \t <from>:<to> \t <summary> (<reason>)
```

The status of up-to-date refs is shown only if --porcelain or --verbose option is used.

flag

A single character indicating the status of the ref:

(space)

for a successfully pushed fast-forward;

`+`

for a successful forced update;

`-`

for a successfully deleted ref;

`*`

for a successfully pushed new ref;

`!`

for a ref that was rejected or failed to push; and

`=`

for a ref that was up to date and did not need pushing.

summary

For a successfully pushed ref, the summary shows the old and new values of the ref in a form suitable for using as an argument to `git log` (this is `<old>..<new>` in most cases, and `<old>...<new>` for forced non-fast-forward updates).

For a failed update, more details are given:

rejected

Git did not try to send the ref at all, typically because it is not a fast-forward and you did not force the update.

remote rejected

The remote end refused the update. Usually caused by a hook on the remote side, or because the remote repository has one of the following safety options in effect: `receive.denyCurrentBranch` (for pushes to the checked out branch), `receive.denyNonFastForwards` (for forced non-fast-forward updates), `receive.denyDeletes` or `receive.denyDeleteCurrent`. See git-config[1].

remote failure

The remote end did not report the successful update of the ref, perhaps because of a temporary error on the remote side, a break in the network connection, or other transient error.

from

The name of the local ref being pushed, minus its `refs/<type>/` prefix. In the case of deletion, the name of the local ref is omitted.

to

The name of the remote ref being updated, minus its `refs/<type>/` prefix.

reason

A human-readable explanation. In the case of successfully pushed refs, no explanation is needed. For a failed ref, the reason for failure is described.

Note about fast-forwards

When an update changes a branch (or more in general, a ref) that used to point at commit A to point at another commit B, it is called a fast-forward update if and only if B is a descendant of A.

In a fast-forward update from A to B, the set of commits that the original commit A built on top of is a subset of the commits the new commit B builds on top of. Hence, it does not lose any history.

In contrast, a non-fast-forward update will lose history. For example, suppose you and somebody else started at the same commit X, and you built a history leading to commit B while the other person built a history leading to commit A. The history looks like this:

```
      B
     /
---X---A
```

Further suppose that the other person already pushed changes leading to A back to the original repository from which you two obtained the original commit X.

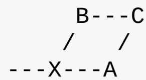
The push done by the other person updated the branch that used to point at commit X to point at commit A. It is a fast-forward.

But if you try to push, you will attempt to update the branch (that now points at A) with commit B. This does *not* fast-forward. If you did so, the changes introduced by commit A will be lost, because everybody will now start building on top of B.

The command by default does not allow an update that is not a fast-forward to prevent such loss of history.

If you do not want to lose your work (history from X to B) or the work by the other person (history from X to A), you would need to first fetch the history from the repository, create a history that contains changes done by both parties, and push the result back.

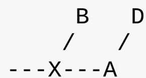
You can perform "git pull", resolve potential conflicts, and "git push" the result. A "git pull" will create a merge commit C between commits A and B.



```
      B---C
     /   /
 ---X---A
```

Updating A with the resulting merge commit will fast-forward and your push will be accepted.

Alternatively, you can rebase your change between X and B on top of A, with "git pull --rebase", and push the result back. The rebase will create a new commit D that builds the change between X and B on top of A.



```
      B   D
     /   /
 ---X---A
```

Again, updating A with this commit will fast-forward and your push will be accepted.

There is another common situation where you may encounter non-fast-forward rejection when you try to push, and it is possible even when you are pushing into a repository nobody else pushes into. After you push commit A yourself (in the first picture in this section), replace it with "git commit --amend" to produce commit B, and you try to push it out, because forgot that you have pushed A out already. In such a case, and only if you are certain that nobody in the meantime fetched your earlier commit A (and started building on top of it), you can run "git push --force" to overwrite it. In other words, "git push --force" is a method reserved for a case where you do mean to lose history.

Examples

`git push`

Works like `git push <remote>`, where <remote> is the current branch’s remote (or `origin`, if no remote is configured for the current branch).

`git push origin`

Without additional configuration, pushes the current branch to the configured upstream (`remote.origin.merge` configuration variable) if it has the same name as the current branch, and errors out without pushing otherwise.

The default behavior of this command when no <refspec> is given can be configured by setting the `push` option of the remote, or the `push.default` configuration variable.

For example, to default to pushing only the current branch to `origin` use `git config remote.origin.push HEAD`. Any valid <refspec> (like the ones in the examples below) can be configured as the default for `git push origin`.

`git push origin :`

Push "matching" branches to `origin`. See <refspec> in the OPTIONS section above for a description of "matching" branches.

`git push origin master`

Find a ref that matches `master` in the source repository (most likely, it would find `refs/heads/master`), and update the same ref (e.g. `refs/heads/master`) in `origin` repository with it. If `master` did not exist remotely, it would be created.

`git push origin HEAD`

A handy way to push the current branch to the same name on the remote.

`git push mothership master:satellite/master dev:satellite/dev`

Use the source ref that matches `master` (e.g. `refs/heads/master`) to update the ref that matches `satellite/master` (most probably `refs/remotes/satellite/master`) in the `mothership` repository; do the same for `dev` and `satellite/dev`.

This is to emulate `git fetch` run on the `mothership` using `git push` that is run in the opposite direction in order to integrate the work done on `satellite`, and is often necessary when you can only make connection in one way (i.e. satellite can ssh into mothership but mothership cannot initiate connection to satellite because the latter is behind a firewall or does not run sshd).

After running this `git push` on the `satellite` machine, you would ssh into the `mothership` and run `git merge` there to complete the emulation of `git pull` that were run on `mothership` to pull changes made on `satellite`.

`git push origin HEAD:master`

Push the current branch to the remote ref matching `master` in the `origin` repository. This form is convenient to push the current branch without thinking about its local name.

`git push origin master:refs/heads/experimental`

Create the branch `experimental` in the `origin` repository by copying the current `master` branch. This form is only needed to create a new branch or tag in the remote repository when the local name and the remote name are different; otherwise, the ref name on its own will work.

`git push origin :experimental`

Find a ref that matches `experimental` in the `origin` repository (e.g. `refs/heads/experimental`), and delete it.

`git push origin +dev:master`

Update the origin repository's master branch with the dev branch, allowing non-fast-forward updates. **This can leave unreferenced commits dangling in the origin repository.** Consider the following situation, where a fast-forward is not possible:

```
          o---o---o---A---B  origin/master
               \
                X---Y---Z  dev
```

The above command would change the origin repository to

```
                A---B  (unnamed branch)
               /
          o---o---o---X---Y---Z  master
```

Commits A and B would no longer belong to a branch with a symbolic name, and so would be unreachable. As such, these commits would be removed by a `git gc` command on the origin repository.

GIT

Part of the git[1] suite

remote

NAME

git-remote - Manage set of tracked repositories

SYNOPSIS

```
git remote [-v | --verbose]
git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>] <na
git remote rename <old> <new>
git remote remove <name>
git remote set-head <name> (-a | --auto | -d | --delete | <branch>)
git remote set-branches [--add] <name> <branch>…
git remote get-url [--push] [--all] <name>
git remote set-url [--push] <name> <newurl> [<oldurl>]
git remote set-url --add [--push] <name> <newurl>
git remote set-url --delete [--push] <name> <url>
git remote [-v | --verbose] show [-n] <name>…
git remote prune [-n | --dry-run] <name>…
git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)…]
```

DESCRIPTION

Manage the set of repositories ("remotes") whose branches you track.

OPTIONS

-v

--verbose

Be a little more verbose and show remote url after name. NOTE: This must be placed between `remote` and `subcommand` .

COMMANDS

With no arguments, shows a list of existing remotes. Several subcommands are available to perform operations on the remotes.

add

Adds a remote named <name> for the repository at <url>. The command `git fetch <name>` can then be used to create and update remote-tracking branches <name>/<branch>.

With `-f` option, `git fetch <name>` is run immediately after the remote information is set up.

With `--tags` option, `git fetch <name>` imports every tag from the remote repository.

With `--no-tags` option, `git fetch <name>` does not import tags from the remote repository.

By default, only tags on fetched branches are imported (see git-fetch[1]).

With `-t <branch>` option, instead of the default glob refspec for the remote to track all branches under the `refs/remotes/<name>/` namespace, a refspec to track only `<branch>` is created. You can give more than one `-t <branch>` to track multiple branches without grabbing all branches.

With `-m <master>` option, a symbolic-ref `refs/remotes/<name>/HEAD` is set up to point at remote's `<master>` branch. See also the set-head command.

When a fetch mirror is created with `--mirror=fetch`, the refs will not be stored in the *refs/remotes/* namespace, but rather everything in *refs/* on the remote will be directly mirrored into *refs/* in the local repository. This option only makes sense in bare repositories, because a fetch would overwrite any local commits.

When a push mirror is created with `--mirror=push`, then `git push` will always behave as if `--mirror` was passed.

rename

Rename the remote named <old> to <new>. All remote-tracking branches and configuration settings for the remote are updated.

In case <old> and <new> are the same, and <old> is a file under `$GIT_DIR/remotes` or `$GIT_DIR/branches`, the remote is converted to the configuration file format.

remove

rm

Remove the remote named <name>. All remote-tracking branches and configuration settings for the remote are removed.

set-head

Sets or deletes the default branch (i.e. the target of the symbolic-ref `refs/remotes/<name>/HEAD`) for the named remote. Having a default branch for a remote is not required, but allows the name of the remote to be specified in lieu of a specific branch. For example, if the default branch for `origin` is set to `master` , then `origin` may be specified wherever you would normally specify `origin/master` .

With `-d` or `--delete` , the symbolic ref `refs/remotes/<name>/HEAD` is deleted.

With `-a` or `--auto` , the remote is queried to determine its `HEAD` , then the symbolic-ref `refs/remotes/<name>/HEAD` is set to the same branch. e.g., if the remote `HEAD` is pointed at `next` , " `git remote set-head origin -a` " will set the symbolic-ref `refs/remotes/origin/HEAD` to `refs/remotes/origin/next` . This will only work if `refs/remotes/origin/next` already exists; if not it must be fetched first.

Use `<branch>` to set the symbolic-ref `refs/remotes/<name>/HEAD` explicitly. e.g., "git remote set-head origin master" will set the symbolic-ref `refs/remotes/origin/HEAD` to `refs/remotes/origin/master` . This will only work if `refs/remotes/origin/master` already exists; if not it must be fetched first.

set-branches

Changes the list of branches tracked by the named remote. This can be used to track a subset of the available remote branches after the initial setup for a remote.

The named branches will be interpreted as if specified with the `-t` option on the *git remote add* command line.

With `--add` , instead of replacing the list of currently tracked branches, adds to that list.

get-url

Retrieves the URLs for a remote. Configurations for `insteadOf` and `pushInsteadOf` are expanded here. By default, only the first URL is listed.

With *--push*, push URLs are queried rather than fetch URLs.

With *--all*, all URLs for the remote will be listed.

set-url

Changes URLs for the remote. Sets first URL for remote <name> that matches regex <oldurl> (first URL if no <oldurl> is given) to <newurl>. If <oldurl> doesn’t match any URL, an error occurs and nothing is changed.

With *--push*, push URLs are manipulated instead of fetch URLs.

With *--add*, instead of changing existing URLs, new URL is added.

With *--delete*, instead of changing existing URLs, all URLs matching regex <url> are deleted for remote <name>. Trying to delete all non-push URLs is an error.

Note that the push URL and the fetch URL, even though they can be set differently, must still refer to the same place. What you pushed to the push URL should be what you would see if you immediately fetched from the fetch URL. If you are trying to fetch from one place (e.g. your upstream) and push to another (e.g. your publishing repository), use two separate remotes.

show

Gives some information about the remote <name>.

With `-n` option, the remote heads are not queried first with `git ls-remote <name>`; cached information is used instead.

prune

Deletes all stale remote-tracking branches under <name>. These stale branches have already been removed from the remote repository referenced by <name>, but are still locally available in "remotes/<name>".

With `--dry-run` option, report what branches will be pruned, but do not actually prune them.

update

Fetch updates for a named set of remotes in the repository as defined by remotes.<group>. If a named group is not specified on the command line, the configuration parameter remotes.default will be used; if remotes.default is not defined, all remotes which do not have the configuration parameter remote.<name>.skipDefaultUpdate set to true will be updated. (See git-config[1]).

With `--prune` option, prune all the remotes that are updated.

DISCUSSION

The remote configuration is achieved using the `remote.origin.url` and `remote.origin.fetch` configuration variables. (See git-config[1]).

Examples

- Add a new remote, fetch, and check out a branch from it

```
$ git remote
origin
$ git branch -r
  origin/HEAD -&gt; origin/master
  origin/master
$ git remote add staging git://git.kernel.org/.../gregkh/staging.git
$ git remote
origin
staging
$ git fetch staging
...
From git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging
 * [new branch]      master     -&gt; staging/master
 * [new branch]      staging-linus -&gt; staging/staging-linus
 * [new branch]      staging-next -&gt; staging/staging-next
$ git branch -r
  origin/HEAD -&gt; origin/master
  origin/master
  staging/master
  staging/staging-linus
  staging/staging-next
$ git checkout -b staging staging/master
...
```

- Imitate *git clone* but track only selected branches

```
$ mkdir project.git
$ cd project.git
$ git init
$ git remote add -f -t master -m master origin git://example.com/git.git/
$ git merge origin
```

SEE ALSO

git-fetch[1] git-branch[1] git-config[1]

GIT

Part of the git[1] suite

submodule

NAME

git-submodule - Initialize, update or inspect submodules

SYNOPSIS

```
git submodule [--quiet] add [-b <branch>] [-f|--force] [--name <name>]
          [--reference <repository>] [--depth <depth>] [--] <repository> [<path>]
git submodule [--quiet] status [--cached] [--recursive] [--] [<path>…]
git submodule [--quiet] init [--] [<path>…]
git submodule [--quiet] deinit [-f|--force] [--] <path>…
git submodule [--quiet] update [--init] [--remote] [-N|--no-fetch]
          [-f|--force] [--rebase|--merge] [--reference <repository>]
          [--depth <depth>] [--recursive] [--] [<path>…]
git submodule [--quiet] summary [--cached|--files] [(-n|--summary-limit) <n>]
          [commit] [--] [<path>…]
git submodule [--quiet] foreach [--recursive] <command>
git submodule [--quiet] sync [--recursive] [--] [<path>…]
```

DESCRIPTION

Inspects, updates and manages submodules.

A submodule allows you to keep another Git repository in a subdirectory of your repository. The other repository has its own history, which does not interfere with the history of the current repository. This can be used to have external dependencies such as third party libraries for example.

When cloning or pulling a repository containing submodules however, these will not be checked out by default; the *init* and *update* subcommands will maintain submodules checked out and at appropriate revision in your working tree.

Submodules are composed from a so-called `gitlink` tree entry in the main repository that refers to a particular commit object within the inner repository that is completely separate. A record in the `.gitmodules` (see gitmodules[5]) file at the root of the source tree assigns a logical name to the submodule and describes the default URL the submodule shall be cloned from. The logical name can be used for overriding this URL within your local repository configuration (see *submodule init*).

Submodules are not to be confused with remotes, which are other repositories of the same project; submodules are meant for different projects you would like to make part of your source tree, while the history of the two projects still stays completely independent and you cannot modify the contents of the submodule from within the main project. If you want to merge the project histories and want to treat the aggregated whole as a single project from then on, you may want to add a remote for the other project and use the *subtree* merge strategy, instead of treating the other project as a submodule. Directories that come from both projects can be cloned and checked out as a whole if you choose to go that route.

COMMANDS

add

Add the given repository as a submodule at the given path to the changeset to be committed next to the current project: the current project is termed the "superproject".

This requires at least one argument: <repository>. The optional argument <path> is the relative location for the cloned submodule to exist in the superproject. If <path> is not given, the "humanish" part of the source repository is used ("repo" for "/path/to/repo.git" and "foo" for "host.xz:foo/.git"). The <path> is also used as the submodule's logical name in its configuration entries unless `--name` is used to specify a logical name.

<repository> is the URL of the new submodule's origin repository. This may be either an absolute URL, or (if it begins with ./ or ../), the location relative to the superproject's origin repository (Please note that to specify a repository *foo.git* which is located right next to a superproject *bar.git*, you'll have to use *../foo.git* instead of *./foo.git* - as one might expect when following the rules for relative URLs - because the evaluation of relative URLs in Git is identical to that of relative directories). If the superproject doesn't have an origin configured the superproject is its own authoritative upstream and the current working directory is used instead.

<path> is the relative location for the cloned submodule to exist in the superproject. If <path> does not exist, then the submodule is created by cloning from the named URL. If <path> does exist and is already a valid Git repository, then this is added to the changeset without cloning. This second form is provided to ease creating a new submodule from scratch, and presumes the user will later push the submodule to the given URL.

In either case, the given URL is recorded into .gitmodules for use by subsequent users cloning the superproject. If the URL is given relative to the superproject's repository, the presumption is the superproject and submodule repositories will be kept together in the same relative location, and only the superproject's URL needs to be provided: git-submodule will correctly locate the submodule using the relative URL in .gitmodules.

status

Show the status of the submodules. This will print the SHA-1 of the currently checked out commit for each submodule, along with the submodule path and the output of *git describe* for the SHA-1. Each SHA-1 will be prefixed with `-` if the submodule is not initialized, `+` if the currently checked out submodule commit does not match the SHA-1 found in the index of the containing repository and `U` if the submodule has merge conflicts.

If `--recursive` is specified, this command will recurse into nested submodules, and show their status as well.

If you are only interested in changes of the currently initialized submodules with respect to the commit recorded in the index or the HEAD, git-status[1] and git-diff[1] will provide that information too (and can also report changes to a submodule's work tree).

init

Initialize the submodules recorded in the index (which were added and committed elsewhere) by copying submodule names and urls from .gitmodules to .git/config. Optional <path> arguments limit which submodules will be initialized. It will also copy the value of `submodule.$name.update` into .git/config. The key used in .git/config is `submodule.$name.url`. This command does not alter existing information in .git/config. You can then customize the submodule clone URLs in .git/config for your local setup and proceed to `git submodule update`; you can also just use `git submodule update --init` without the explicit *init* step if you do not intend to customize any submodule locations.

deinit

Unregister the given submodules, i.e. remove the whole `submodule.$name` section from .git/config together with their work tree. Further calls to `git submodule update`, `git submodule foreach` and `git submodule sync` will skip any unregistered submodules until they are initialized again, so use this command if you don't want to have a local checkout of the submodule in your work tree anymore. If you really want to remove a submodule from the repository and commit that use git-rm[1] instead.

If `--force` is specified, the submodule's work tree will be removed even if it contains local modifications.

update

Update the registered submodules to match what the superproject expects by cloning missing submodules and updating the working tree of the submodules. The "updating" can be done in several ways depending on command line options and the value of `submodule.<name>.update` configuration variable. Supported update procedures are:

checkout

the commit recorded in the superproject will be checked out in the submodule on a detached HEAD. This is done when `--checkout` option is given, or no option is given, and `submodule.<name>.update` is unset, or if it is set to *checkout*.

If `--force` is specified, the submodule will be checked out (using `git checkout --force` if appropriate), even if the commit specified in the index of the containing repository already matches the commit checked out in the submodule.

rebase

the current branch of the submodule will be rebased onto the commit recorded in the superproject. This is done when `--rebase` option is given, or no option is given, and `submodule.<name>.update` is set to *rebase*.

merge

the commit recorded in the superproject will be merged into the current branch in the submodule. This is done when `--merge` option is given, or no option is given, and `submodule.<name>.update` is set to *merge*.

custom command

arbitrary shell command that takes a single argument (the sha1 of the commit recorded in the superproject) is executed. This is done when no option is given, and `submodule.<name>.update` has the form of *!command*.

When no option is given and `submodule.<name>.update` is set to *none*, the submodule is not updated.

If the submodule is not yet initialized, and you just want to use the setting as stored in .gitmodules, you can automatically initialize the submodule with the `--init` option.

If `--recursive` is specified, this command will recurse into the registered submodules, and update any nested submodules within.

summary

Show commit summary between the given commit (defaults to HEAD) and working tree/index. For a submodule in question, a series of commits in the submodule between the given super project commit and the index or working tree (switched by `--cached`) are shown. If the option `--files` is given, show the series of commits in the submodule between the index of the super project and the working tree of the submodule (this option doesn’t allow to use the `--cached` option or to provide an explicit commit).

Using the `--submodule=log` option with git-diff[1] will provide that information too.

foreach

Evaluates an arbitrary shell command in each checked out submodule. The command has access to the variables $name, $path, $sha1 and $toplevel: $name is the name of the relevant submodule section in .gitmodules, $path is the name of the submodule directory relative to the superproject, $sha1 is the commit as recorded in the superproject, and $toplevel is the absolute path to the top-level of the superproject. Any submodules defined in the superproject but not checked out are ignored by this command. Unless given `--quiet`, foreach prints the name of each submodule before evaluating the command. If `--recursive` is given, submodules are traversed recursively (i.e. the given shell command is evaluated in nested submodules as well). A non-zero return from the command in any submodule causes the processing to terminate. This can be overridden by adding || *:* to the end of the command.

As an example, `git submodule foreach 'echo $path` git rev-parse HEAD'` will show the path and currently checked out commit for each submodule.

sync

Synchronizes submodules' remote URL configuration setting to the value specified in .gitmodules. It will only affect those submodules which already have a URL entry in .git/config (that is the case when they are initialized or freshly added). This is useful when submodule URLs change upstream and you need to update your local repositories accordingly.

"git submodule sync" synchronizes all submodules while "git submodule sync -- A" synchronizes submodule "A" only.

If `--recursive` is specified, this command will recurse into the registered submodules, and sync any nested submodules within.

OPTIONS

-q

--quiet

Only print error messages.

-b

--branch

Branch of repository to add as submodule. The name of the branch is recorded as `submodule.<name>.branch` in `.gitmodules` for `update --remote`.

-f

--force

This option is only valid for add, deinit and update commands. When running add, allow adding an otherwise ignored submodule path. When running deinit the submodule work trees will be removed even if they contain local changes. When running update (only effective with the checkout procedure), throw away local changes in submodules when switching to a different commit; and always run a checkout operation in the submodule, even if the commit listed in the index of the containing repository matches the commit checked out in the submodule.

--cached

This option is only valid for status and summary commands. These commands typically use the commit found in the submodule HEAD, but with this option, the commit stored in the index is used instead.

--files

This option is only valid for the summary command. This command compares the commit in the index with that in the submodule HEAD when this option is used.

-n

--summary-limit

This option is only valid for the summary command. Limit the summary size (number of commits shown in total). Giving 0 will disable the summary; a negative number means unlimited (the default). This limit only applies to modified submodules. The size is always limited to 1 for added/deleted/typechanged submodules.

--remote

This option is only valid for the update command. Instead of using the superproject's recorded SHA-1 to update the submodule, use the status of the submodule's remote-tracking branch. The remote used is branch's remote (`branch.<name>.remote`), defaulting to `origin` . The remote branch used defaults to `master` , but the branch name may be overridden by setting the `submodule.<name>.branch` option in either `.gitmodules` or `.git/config` (with `.git/config` taking precedence).

This works for any of the supported update procedures (`--checkout` , `--rebase` , etc.). The only change is the source of the target SHA-1. For example, `submodule update --remote --merge` will merge upstream submodule changes into the submodules, while `submodule update --merge` will merge superproject gitlink changes into the submodules.

In order to ensure a current tracking branch state, `update --remote` fetches the submodule's remote repository before calculating the SHA-1. If you don't want to fetch, you should use `submodule update --remote --no-fetch`.

Use this option to integrate changes from the upstream subproject with your submodule's current HEAD. Alternatively, you can run `git pull` from the submodule, which is equivalent except for the remote branch name: `update --remote` uses the default upstream repository and `submodule.<name>.branch`, while `git pull` uses the submodule's `branch.<name>.merge`. Prefer `submodule.<name>.branch` if you want to distribute the default upstream branch with the superproject and `branch.<name>.merge` if you want a more native feel while working in the submodule itself.

-N

--no-fetch

This option is only valid for the update command. Don't fetch new objects from the remote site.

--checkout

This option is only valid for the update command. Checkout the commit recorded in the superproject on a detached HEAD in the submodule. This is the default behavior, the main use of this option is to override `submodule.$name.update` when set to a value other than `checkout`. If the key `submodule.$name.update` is either not explicitly set or set to `checkout`, this option is implicit.

--merge

This option is only valid for the update command. Merge the commit recorded in the superproject into the current branch of the submodule. If this option is given, the submodule's HEAD will not be detached. If a merge failure prevents this process, you will have to resolve the resulting conflicts within the submodule with the usual conflict resolution tools. If the key `submodule.$name.update` is set to `merge`, this option is implicit.

--rebase

This option is only valid for the update command. Rebase the current branch onto the commit recorded in the superproject. If this option is given, the submodule's HEAD will not be detached. If a merge failure prevents this process, you will have to resolve these failures with git-rebase[1]. If the key `submodule.$name.update` is set to `rebase`, this option is implicit.

--init

This option is only valid for the update command. Initialize all submodules for which "git submodule init" has not been called so far before updating.

--name

This option is only valid for the add command. It sets the submodule’s name to the given string instead of defaulting to its path. The name must be valid as a directory name and may not end with a /.

--reference <repository>

This option is only valid for add and update commands. These commands sometimes need to clone a remote repository. In this case, this option will be passed to the git-clone[1] command.

NOTE: Do **not** use this option unless you have read the note for git-clone[1]'s `--reference` and `--shared` options carefully.

--recursive

This option is only valid for foreach, update, status and sync commands. Traverse submodules recursively. The operation is performed not only in the submodules of the current repo, but also in any nested submodules inside those submodules (and so on).

--depth

This option is valid for add and update commands. Create a *shallow* clone with a history truncated to the specified number of revisions. See git-clone[1]

<path>…

Paths to submodule(s). When specified this will restrict the command to only operate on the submodules found at the specified paths. (This argument is required with add).

FILES

When initializing submodules, a .gitmodules file in the top-level directory of the containing repository is used to find the url of each submodule. This file should be formatted in the same way as `$GIT_DIR/config`. The key to each submodule url is "submodule.$name.url". See gitmodules[5] for details.

GIT

Part of the git[1] suite

Inspection and Comparison

show

NAME

git-show - Show various types of objects

SYNOPSIS

```
git show [options] <object>…
```

DESCRIPTION

Shows one or more objects (blobs, trees, tags and commits).

For commits it shows the log message and textual diff. It also presents the merge commit in a special format as produced by *git diff-tree --cc*.

For tags, it shows the tag message and the referenced objects.

For trees, it shows the names (equivalent to *git ls-tree* with --name-only).

For plain blobs, it shows the plain contents.

The command takes options applicable to the *git diff-tree* command to control how the changes the commit introduces are shown.

This manual page describes only the most frequently used options.

OPTIONS

<object>…

The names of objects to show. For a more complete list of ways to spell object names, see "SPECIFYING REVISIONS" section in gitrevisions[7].

--pretty[=<format>]

--format=<format>

Pretty-print the contents of the commit logs in a given format, where *<format>* can be one of *oneline*, *short*, *medium*, *full*, *fuller*, *email*, *raw*, *format:<string>* and *tformat:<string>*. When *<format>* is none of the above, and has *%placeholder* in it, it acts as if *--pretty=tformat:<format>* were given.

See the "PRETTY FORMATS" section for some additional details for each format. When = *<format>* part is omitted, it defaults to *medium*.

Note: you can specify the default pretty format in the repository configuration (see git-config[1]).

--abbrev-commit

Instead of showing the full 40-byte hexadecimal commit object name, show only a partial prefix. Non default number of digits can be specified with "--abbrev=<n>" (which also modifies diff output, if it is displayed).

This should make "--pretty=oneline" a whole lot more readable for people using 80-column terminals.

--no-abbrev-commit

Show the full 40-byte hexadecimal commit object name. This negates `--abbrev-commit` and those options which imply it such as "--oneline". It also overrides the *log.abbrevCommit* variable.

--oneline

This is a shorthand for "--pretty=oneline --abbrev-commit" used together.

--encoding=<encoding>

The commit objects record the encoding used for the log message in their encoding header; this option can be used to tell the command to re-code the commit log message in the encoding preferred by the user. For non plumbing commands this defaults to UTF-8. Note that if an object claims to be encoded in `X` and we are outputting in `X`, we will output the object verbatim; this means that invalid sequences in the original commit may be copied to the output.

--notes[=<treeish>]

Show the notes (see git-notes[1]) that annotate the commit, when showing the commit log message. This is the default for `git log`, `git show` and `git whatchanged` commands when there is no `--pretty`, `--format`, or `--oneline` option given on the command line.

By default, the notes shown are from the notes refs listed in the *core.notesRef* and *notes.displayRef* variables (or corresponding environment overrides). See git-config[1] for more details.

With an optional *<treeish>* argument, use the treeish to find the notes to display. The treeish can specify the full refname when it begins with `refs/notes/`; when it begins with `notes/`, `refs/` and otherwise `refs/notes/` is prefixed to form a full name of the ref.

Multiple --notes options can be combined to control which notes are being displayed. Examples: "--notes=foo" will show only notes from "refs/notes/foo"; "--notes=foo --notes" will show both notes from "refs/notes/foo" and from the default notes ref(s).

--no-notes

Do not show notes. This negates the above `--notes` option, by resetting the list of notes refs from which notes are shown. Options are parsed in the order given on the command line, so e.g. "--notes --notes=foo --no-notes --notes=bar" will only show notes from "refs/notes/bar".

--show-notes[=<treeish>]

--[no-]standard-notes

These options are deprecated. Use the above --notes/--no-notes options instead.

--show-signature

Check the validity of a signed commit object by passing the signature to `gpg --verify` and show the output.

PRETTY FORMATS

If the commit is a merge, and if the pretty-format is not *oneline*, *email* or *raw*, an additional line is inserted before the *Author:* line. This line begins with "Merge: " and the sha1s of ancestral commits are printed, separated by spaces. Note that the listed commits may not necessarily be the list of the **direct** parent commits if you have limited your view of history: for example, if you are only interested in changes related to a certain directory or file.

There are several built-in formats, and you can define additional formats by setting a pretty.<name> config option to either another format name, or a *format:* string, as described below (see git-config[1]). Here are the details of the built-in formats:

- *oneline*

  ```
  &lt;sha1&gt; &lt;title line&gt;
  ```

This is designed to be as compact as possible.

- *short*

```
commit &lt;sha1&gt;
Author: &lt;author&gt;
```

```
&lt;title line&gt;
```

- *medium*

```
commit &lt;sha1&gt;
Author: &lt;author&gt;
Date:   &lt;author date&gt;
```

```
&lt;title line&gt;
```

```
&lt;full commit message&gt;
```

- *full*

```
commit &lt;sha1&gt;
Author: &lt;author&gt;
Commit: &lt;committer&gt;
```

```
&lt;title line&gt;
```

```
&lt;full commit message&gt;
```

- *fuller*

```
commit &lt;sha1&gt;
Author:     &lt;author&gt;
AuthorDate: &lt;author date&gt;
Commit:     &lt;committer&gt;
CommitDate: &lt;committer date&gt;
```

```
&lt;title line&gt;
```

```
&lt;full commit message&gt;
```

- *email*

```
From &lt;sha1&gt; &lt;date&gt;
From: &lt;author&gt;
Date: &lt;author date&gt;
Subject: [PATCH] &lt;title line&gt;
```

```
&lt;full commit message&gt;
```

- *raw*

  The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the SHA-1s are displayed in full, regardless of whether --abbrev or --no-abbrev are used, and *parents* information show the true parent commits, without taking grafts or history simplification into account. Note that this format affects the way commits are displayed, but not the way the diff is shown e.g. with `git log --raw`. To get full object names in a raw diff format, use `--no-abbrev`.

- *format:<string>*

  The *format:<string>* format allows you to specify which information you want to show. It works a little bit like printf format, with the notable exception that you get a newline with *%n* instead of *\n*.

  E.g, *format:"The author of %h was %an, %ar%nThe title was >>%s<<%n"* would show something like this:

```
The author of fe6e0ee was Junio C Hamano, 23 hours ago
The title was &gt;&gt;t4119: test autocomputing -p&lt;n&gt; for traditional diff inpu
```

  The placeholders are:

  - *%H*: commit hash
  - *%h*: abbreviated commit hash
  - *%T*: tree hash
  - *%t*: abbreviated tree hash
  - *%P*: parent hashes
  - *%p*: abbreviated parent hashes
  - *%an*: author name
  - *%aN*: author name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
  - *%ae*: author email

- *%aE*: author email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ad*: author date (format respects --date= option)
- *%aD*: author date, RFC2822 style
- *%ar*: author date, relative
- *%at*: author date, UNIX timestamp
- *%ai*: author date, ISO 8601-like format
- *%aI*: author date, strict ISO 8601 format
- *%cn*: committer name
- *%cN*: committer name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ce*: committer email
- *%cE*: committer email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%cd*: committer date (format respects --date= option)
- *%cD*: committer date, RFC2822 style
- *%cr*: committer date, relative
- *%ct*: committer date, UNIX timestamp
- *%ci*: committer date, ISO 8601-like format
- *%cI*: committer date, strict ISO 8601 format
- *%d*: ref names, like the --decorate option of git-log[1]
- *%D*: ref names without the " (", ")" wrapping.
- *%e*: encoding
- *%s*: subject
- *%f*: sanitized subject line, suitable for a filename
- *%b*: body
- *%B*: raw body (unwrapped subject and body)
- *%N*: commit notes
- *%GG*: raw verification message from GPG for a signed commit

- *%G?*: show "G" for a Good signature, "B" for a Bad signature, "U" for a good, untrusted signature and "N" for no signature
- *%GS*: show the name of the signer for a signed commit
- *%GK*: show the key used to sign a signed commit
- *%gD*: reflog selector, e.g., `refs/stash@{1}`
- *%gd*: shortened reflog selector, e.g., `stash@{1}`
- *%gn*: reflog identity name
- *%gN*: reflog identity name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ge*: reflog identity email
- *%gE*: reflog identity email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%gs*: reflog subject
- *%Cred*: switch color to red
- *%Cgreen*: switch color to green
- *%Cblue*: switch color to blue
- *%Creset*: reset color
- *%C(…)*: color specification, as described in color.branch.* config option; adding `auto,` at the beginning will emit color only when colors are enabled for log output (by `color.diff`, `color.ui`, or `--color`, and respecting the `auto` settings of the former if we are going to a terminal). `auto` alone (i.e. `%C(auto)`) will turn on auto coloring on the next placeholders until the color is switched again.
- *%m*: left, right or boundary mark
- *%n*: newline
- *%%*: a raw *%*
- *%x00*: print a byte from a hex code
- *%w([<w>[,<i1>[,<i2>]]])*: switch line wrapping, like the -w option of git-shortlog[1].
- *%<(<N>[,trunc|ltrunc|mtrunc])*: make the next placeholder take at least N columns, padding spaces on the right if necessary. Optionally truncate at the beginning (ltrunc), the middle (mtrunc) or the end (trunc) if the output is longer than N columns. Note that truncating only works correctly with N >= 2.

- *%<|(<N>)*: make the next placeholder take at least until Nth columns, padding spaces on the right if necessary
- *%>(<N>)*, *%>|(<N>)*: similar to *%<(<N>)*, *%<|(<N>)* respectively, but padding spaces on the left
- *%>>(<N>)*, *%>>|(<N>)*: similar to *%>(<N>)*, *%>|(<N>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces
- *%><(<N>)*, *%><|(<N>)*: similar to *% <(<N>)*, *%<|(<N>)* respectively, but padding both sides (i.e. the text is centered)

> Note
>
> Some placeholders may depend on other options given to the revision traversal engine. For example, the `%g*` reflog options will insert an empty string unless we are traversing reflog entries (e.g., by `git log -g`). The `%d` and `%D` placeholders will use the "short" decoration format if `--decorate` was not already provided on the command line.

If you add a `+` (plus sign) after *%* of a placeholder, a line-feed is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

If you add a `-` (minus sign) after *%* of a placeholder, line-feeds that immediately precede the expansion are deleted if and only if the placeholder expands to an empty string.

If you add a ` ` (space) after *%* of a placeholder, a space is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

- *tformat:*

  The *tformat:* format works exactly like *format:*, except that it provides "terminator" semantics instead of "separator" semantics. In other words, each commit has the message terminator character (usually a newline) appended, rather than a separator placed between entries. This means that the final entry of a single-line format will be properly terminated with a new line, just as the "oneline" format does. For example:

  ```
  $ git log -2 --pretty=format:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973 -- NO NEWLINE

  $ git log -2 --pretty=tformat:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973
  ```

In addition, any unrecognized string that has a `%` in it is interpreted as if it has `tformat:` in front of it. For example, these two are equivalent:

```
$ git log -2 --pretty=tformat:%h 4da45bef
$ git log -2 --pretty=%h 4da45bef
```

COMMON DIFF OPTIONS

-p

-u

--patch

Generate patch (see section on generating patches).

-s

--no-patch

Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw

For each commit, show a summary of changes using the raw diff format. See the "RAW OUTPUT FORMAT" section of git-diff[1]. This is different from showing the log itself in raw format, which you can achieve with `--format=raw`.

--patch-with-raw

Synonym for `-p --raw`.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

`default` , `myers`

The basic greedy diff algorithm. Currently, this is the default.

`minimal`

Spend extra time to make sure the smallest possible diff is produced.

`patience`

Use "patience diff" algorithm when generating patches.

`histogram`

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured diff.algorithm variable to a non-default value and want to use the default one, then you have to use `--diff-algorithm=default` option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by `<width>`. The width of the filename part can be limited by giving another width `<name-width>` after a comma. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (does not affect `git format-patch`). By giving a third parameter `<count>`, you can limit the output to the first `<count>` lines, followed by `...` if there are more.

These parameters can also be set individually with `--stat-width=<width>`, `--stat-name-width=<name-width>` and `--stat-count=<count>`.

--numstat

Similar to `--stat`, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two `-` instead of saying `0 0`.

--shortstat

Output only the last line of the `--stat` format containing total number of modified files, as well as number of added and deleted lines.

--dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of `--dirstat` can be customized by passing it a comma separated list of parameters. The defaults are controlled by the `diff.dirstat` configuration variable (see git-config[1]). The following parameters are available:

`changes`

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

`lines`

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive `--dirstat` behavior than the `changes` behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other `--*stat` options.

`files`

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest `--dirstat` behavior, since it does not have to look at the file contents at all.

`cumulative`

Count changes in a child directory for the parent directory as well. Note that when using `cumulative`, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the `noncumulative` parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: `--dirstat=files,10,cumulative`.

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for `-p --stat`.

-z

Separate the commits with NULs instead of with new newlines.

Also, when `--raw` or `--numstat` has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only

Show only names of changed files.

--name-status

Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]

Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule[1] `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--color[=<when>]

Show colored diff. `--color` (i.e. without =<*when*>) is the same as `--color=always`. <*when*> can be one of `always`, `never`, or `auto`.

--no-color

Turn off colored diff. It is the same as `--color=never`.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see `--word-diff-regex` below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies `--color`.

plain

Show words as `[-removed-]` and `{+added+}`. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a `+` / `-` / `` ` character at the beginning of the line and extending to the end of the line. Newl``
~` on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies `--word-diff` unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append `|[^[:space:]]` to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, `--word-diff-regex=.` will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see gitattributes[1] or git-config[1]. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to `--word-diff=color` plus (if a regex was specified) `--word-diff-regex=<regex>`.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--check

Warn if changes introduce whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors on lines specified by <kind> in the color specified by `color.diff.whitespace`. <kind> is a comma separated list of `old`, `new`, `context`. When this option is not given, only whitespace errors in `new` lines are highlighted. E.g. `--ws-error-highlight=new,old` highlights whitespace errors on both deleted and added lines. `all` can be used as a short-hand for `old,new,context`.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to `--full-index`, output a binary diff that can be applied with `git-apply`.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. Non default number of digits can be specified with `--abbrev=<n>`.

-B[<n>][/<m>]

--break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything

new, and the number `m` controls this aspect of the -B option (defaults to 60%). `-B/70%` specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number `n` controls this aspect of the -B option (defaults to 50%). `-B20%` specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>]

--find-renames[=<n>]

If generating diffs, detect and report renames for each commit. For following files across renames while traversing history, see `--follow`. If `n` is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, `-M90%` means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a `%` sign, the number is to be read as a fraction, with a decimal point before it. I.e., `-M5` becomes 0.5, and is thus the same as `-M50%`. Similarly, `-M05` is the same as `-M5%`. To limit detection to exact renames, use `-M100%`. The default similarity index is 50%.

-C[<n>]

--find-copies[=<n>]

Detect copies as well as renames. See also `--find-copies-harder`. If `n` is specified, it has the same meaning as for `-M<n>`.

--find-copies-harder

For performance reasons, by default, `-C` option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one `-C` option has the same effect.

-D

--irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and `/dev/null`. The resulting patch is not meant to be applied with `patch` or `git apply`; this is solely for people who want to just concentrate on reviewing the text after

the change. In addition, the output obviously lack enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with `-B`, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The `-M` and `-C` options require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (`A`), Copied (`C`), Deleted (`D`), Modified (`M`), Renamed (`R`), have their type (i.e. regular file, symlink, submodule, …) changed (`T`), are Unmerged (`U`), are Unknown (`X`), or have had their pairing Broken (`B`). Any combination of the filter characters (including none) can be used. When `*` (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter's use.

It is useful when you're looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into `-S`, and keep going until you get the very first version of the block.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between `-S<regex> --pickaxe-regex` and `-G<regex>`, consider a commit with the following diff in the same file:

```
+    return !regexec(regexp, two-&gt;ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

See the *pickaxe* entry in gitdiffcore[7] for more information.

--pickaxe-all

When `-S` or `-G` finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to `-S` as an extended POSIX regular expression to match.

-O<orderfile>

Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config[1]). To cancel `diff.orderFile`, use `-O/dev/null`.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument.

-a

--text

Treat all files as text.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b

--ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w

--ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other.

-W

--function-context

Show whole surrounding functions of changes.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with gitattributes[5], you need to use this option with git-log[1] and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv

--no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See gitattributes[5] for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for git-diff[1] and git-log[1], but not for git-format-patch[1] or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in git-config[1] or gitmodules[5]. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

For more detailed explanation on these common options, see also gitdiffcore[7].

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The `a/` and `b/` filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, `/dev/null` is *not* used in place of the `a/` or `b/` filenames.

   When rename/copy is involved, `file1` and `file2` show the name of the source file of the rename/copy and the name of the file that rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

   ```
   old mode &lt;mode&gt;
   new mode &lt;mode&gt;
   deleted file mode &lt;mode&gt;
   new file mode &lt;mode&gt;
   copy from &lt;path&gt;
   copy to &lt;path&gt;
   rename from &lt;path&gt;
   rename to &lt;path&gt;
   similarity index &lt;number&gt;
   dissimilarity index &lt;number&gt;
   index &lt;hash&gt;..&lt;hash&gt; &lt;mode&gt;
   ```

   File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the `a/` and `b/` prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the SHA-1 checksum before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

4. All the `file1` files in the output refer to files before the commit, and all the `file2` files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

combined diff format

Any diff-generating command can take the `-c` or `--cc` option to produce a *combined diff* when showing a merge. This is the default format when showing merges with git-diff[1] or git-show[1]. Note also that you can give the `-m` option to any of these commands to force generation of diffs with individual parents of a merge.

A *combined diff* format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
    return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +    unsigned char sha1[20];
 +    struct commit *cmit;
    struct commit_list *list;
    static int initialized = 0;
    struct commit_name *n;

 +    if (get_sha1(arg, sha1) < 0)
 +        usage(describe_usage);
 +    cmit = lookup_commit_reference(sha1);
 +    if (!cmit)
 +        usage(describe_usage);
 +
    if (!initialized) {
        initialized = 1;
        for_each_ref(get_name);
```

1. It is preceded with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index &lt;hash&gt;,&lt;hash&gt;..&lt;hash&gt;
   mode &lt;mode&gt;,&lt;mode&gt;..&lt;mode&gt;
   new file mode &lt;mode&gt;
   deleted file mode &lt;mode&gt;,&lt;mode&gt;
   ```

   The `mode <mode>,<mode>..<mode>` line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

   Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.

4. Chunk header format is modified to prevent people from accidentally feeding it to `patch -p1`. Combined diff format was created for review of merge commit changes, and was not meant for apply. The change is similar to the change in the extended *index* header:

   ```
   @@@ &lt;from-file-range&gt; &lt;from-file-range&gt; &lt;to-file-range&gt; @@@
   ```

   There are (number of parents + 1) `@` characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has `-` (minus — appears in A but removed in B), `+` (plus — missing in A but added to B), or `" "` (space — unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A `-` character in the column N means that the line appears in fileN but it does not appear in the result. A `+` character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two `-` removals from both file1 and file2, plus `++` to mean one line that was added does not appear in either file1 or file2). Also eight other lines are the same from file1 but do not appear in file2 (hence prefixed with `+`).

When shown by `git diff-tree -c`, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by `git diff-files -c`, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

EXAMPLES

`git show v1.0.0`

Shows the tag `v1.0.0`, along with the object the tags points at.

`git show v1.0.0^{tree}`

Shows the tree pointed to by the tag `v1.0.0`.

`git show -s --format=%s v1.0.0^{commit}`

Shows the subject of the commit pointed to by the tag `v1.0.0`.

`git show next~10:Documentation/README`

Shows the contents of the file `Documentation/README` as they were current in the 10th last commit of the branch `next`.

`git show master:Makefile master:t/Makefile`

Concatenates the contents of said Makefiles in the head of the branch `master`.

Discussion

Git is to some extent character encoding agnostic.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (`.git/config` (see git-config[1]), gitignore[5], gitattributes[5] and gitmodules[5]).

  Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that use legacy extended ASCII encodings. However, repositories created on such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.
- Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but *not* UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

Although we encourage that the commit log messages are encoded in UTF-8, both the core and Git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, Git does not forbid it. However, there are a few things to keep in mind.

1. *git commit* and *git commit-tree* issues a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have i18n.commitencoding in `.git/config` file, like this:

```
[i18n]
    commitencoding = ISO-8859-1
```

Commit objects created with the above setting record the value of `i18n.commitencoding` in its `encoding` header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.

2. *git log*, *git show*, *git blame* and friends look at the `encoding` header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with `i18n.logoutputencoding` in `.git/config` file, like this:

```
[i18n]
    logoutputencoding = ISO-8859-1
```

If you do not have this configuration variable, the value of `i18n.commitencoding` is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

GIT

Part of the git[1] suite

log

NAME

git-log - Show commit logs

SYNOPSIS

```
git log [<options>] [<revision range>] [[\--] <path>…]
```

DESCRIPTION

Shows the commit logs.

The command takes options applicable to the `git rev-list` command to control what is shown and how, and options applicable to the `git diff-*` commands to control how the changes each commit introduces are shown.

OPTIONS

--follow

Continue listing the history of a file beyond renames (works only for a single file).

--no-decorate

--decorate[=short|full|no]

Print out the ref names of any commits that are shown. If *short* is specified, the ref name prefixes *refs/heads/*, *refs/tags/* and *refs/remotes/* will not be printed. If *full* is specified, the full ref name (including prefix) will be printed. The default option is *short*.

--source

Print out the ref name given on the command line by which each commit was reached.

--use-mailmap

Use mailmap file to map author and committer names and email addresses to canonical real names and email addresses. See git-shortlog[1].

--full-diff

Without this flag, `git log -p <path>...` shows commits that touch the specified paths, and diffs about the same specified paths. With this, the full diff is shown for commits that touch the specified paths; this means that "<path>…" limits only commits, and doesn’t limit diff for those commits.

Note that this affects all diff-based output types, e.g. those produced by `--stat`, etc.

--log-size

Include a line “log size <number>” in the output for each commit, where <number> is the length of that commit’s message in bytes. Intended to speed up tools that read log messages from `git log` output by allowing them to allocate space in advance.

-L <start>,<end>:<file>

-L :<funcname>:<file>

Trace the evolution of the line range given by "<start>,<end>" (or the function name regex <funcname>) within the <file>. You may not give any pathspec limiters. This is currently limited to a walk starting from a single revision, i.e., you may only give zero or one positive revision arguments. You can specify this option more than once.

<start> and <end> can take one of these forms:

- number

  If <start> or <end> is a number, it specifies an absolute line number (lines count from 1).

- /regex/

  This form will use the first line matching the given POSIX regex. If <start> is a regex, it will search from the end of the previous `-L` range, if any, otherwise from the start of file. If <start> is “^/regex/”, it will search from the start of file. If <end> is a regex, it will search starting at the line given by <start>.

- +offset or -offset

  This is only valid for <end> and will specify a number of lines before or after the line given by <start>.

If “:<funcname>” is given in place of <start> and <end>, it is a regular expression that denotes the range from the first funcname line that matches <funcname>, up to the next funcname line. “:<funcname>” searches from the end of the previous `-L` range, if any, otherwise from the start of file. “^:<funcname>” searches from the start of file.

<revision range>

Show only commits in the specified revision range. When no <revision range> is specified, it defaults to `HEAD` (i.e. the whole history leading to the current commit). `origin..HEAD` specifies all the commits reachable from the current commit (i.e. `HEAD`), but not from `origin`. For a complete list of ways to spell <revision range>, see the *Specifying Ranges* section of gitrevisions[7].

[--] <path>…

Show only commits that are enough to explain how the files that match the specified paths came to be. See *History Simplification* below for details and other simplification modes.

Paths may need to be prefixed with ‘`-- ’’ to separate them from options or the revision range, when confusion arises.

Commit Limiting

Besides specifying a range of commits that should be listed using the special notations explained in the description, additional commit limiting may be applied.

Using more options generally further limits the output (e.g. `--since=<date1>` limits to commits newer than `<date1>`, and using it with `--grep=<pattern>` further limits to commits whose log message has a line that matches `<pattern>`), unless otherwise noted.

Note that these are applied before commit ordering and formatting options, such as `--reverse`.

-<number>

-n <number>

--max-count=<number>

Limit the number of commits to output.

--skip=<number>

Skip *number* commits before starting to show the commit output.

--since=<date>

--after=<date>

Show commits more recent than a specific date.

--until=<date>

--before=<date>

Show commits older than a specific date.

--author=<pattern>

--committer=<pattern>

Limit the commits output to ones with author/committer header lines that match the specified pattern (regular expression). With more than one `--author=<pattern>`, commits whose author matches any of the given patterns are chosen (similarly for multiple `--committer=<pattern>`).

--grep-reflog=<pattern>

Limit the commits output to ones with reflog entries that match the specified pattern (regular expression). With more than one `--grep-reflog`, commits whose reflog message matches any of the given patterns are chosen. It is an error to use this option unless `--walk-reflogs` is in use.

--grep=<pattern>

Limit the commits output to ones with log message that matches the specified pattern (regular expression). With more than one `--grep=<pattern>`, commits whose message matches any of the given patterns are chosen (but see `--all-match`).

When `--show-notes` is in effect, the message from the notes is matched as if it were part of the log message.

--all-match

Limit the commits output to ones that match all given `--grep`, instead of ones that match at least one.

--invert-grep

Limit the commits output to ones with log message that do not match the pattern specified with `--grep=<pattern>`.

-i

--regexp-ignore-case

Match the regular expression limiting patterns without regard to letter case.

--basic-regexp

Consider the limiting patterns to be basic regular expressions; this is the default.

-E

--extended-regexp

Consider the limiting patterns to be extended regular expressions instead of the default basic regular expressions.

-F

--fixed-strings

Consider the limiting patterns to be fixed strings (don’t interpret pattern as a regular expression).

--perl-regexp

Consider the limiting patterns to be Perl-compatible regular expressions. Requires libpcre to be compiled in.

--remove-empty

Stop when a given path disappears from the tree.

--merges

Print only merge commits. This is exactly the same as `--min-parents=2` .

--no-merges

Do not print commits with more than one parent. This is exactly the same as `--max-parents=1` .

--min-parents=<number>

--max-parents=<number>

--no-min-parents

--no-max-parents

Show only commits which have at least (or at most) that many parent commits. In particular, `--max-parents=1` is the same as `--no-merges` , `--min-parents=2` is the same as `--merges` . `--max-parents=0` gives all root commits and `--min-parents=3` all octopus merges.

`--no-min-parents` and `--no-max-parents` reset these limits (to no limit) again. Equivalent forms are `--min-parents=0` (any commit has 0 or more parents) and `--max-parents=-1` (negative numbers denote no upper limit).

--first-parent

Follow only the first parent commit upon seeing a merge commit. This option can give a better overview when viewing the evolution of a particular topic branch, because merges into a topic branch tend to be only about adjusting to updated upstream from time to time, and this option allows you to ignore the individual commits brought in to your history by such a merge. Cannot be combined with --bisect.

--not

Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next `--not` .

--all

Pretend as if all the refs in `refs/` are listed on the command line as <*commit*>.

--branches[=<pattern>]

Pretend as if all the refs in `refs/heads` are listed on the command line as <*commit*>. If <*pattern*> is given, limit branches to ones matching given shell glob. If pattern lacks *?*, *, or *[*, */** at the end is implied.

--tags[=<pattern>]

Pretend as if all the refs in `refs/tags` are listed on the command line as <*commit*>. If <*pattern*> is given, limit tags to ones matching given shell glob. If pattern lacks *?*, *, or *[*, */** at the end is implied.

--remotes[=<pattern>]

Pretend as if all the refs in `refs/remotes` are listed on the command line as <*commit*>. If <*pattern*> is given, limit remote-tracking branches to ones matching given shell glob. If pattern lacks *?*, *, or *[*, */** at the end is implied.

--glob=<glob-pattern>

Pretend as if all the refs matching shell glob <*glob-pattern*> are listed on the command line as <*commit*>. Leading *refs/*, is automatically prepended if missing. If pattern lacks *?*, *, or *[*, */** at the end is implied.

--exclude=<glob-pattern>

Do not include refs matching <*glob-pattern*> that the next `--all` , `--branches` , `--tags` , `--remotes` , or `--glob` would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next `--all` , `--branches` , `--tags` , `--remotes` , or `--glob` option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with `refs/heads`, `refs/tags`, or `refs/remotes` when applied to `--branches`, `--tags`, or `--remotes`, respectively, and they must begin with `refs/` when applied to `--glob` or `--all`. If a trailing /* is intended, it must be given explicitly.

--reflog

Pretend as if all objects mentioned by reflogs are listed on the command line as `<commit>`.

--ignore-missing

Upon seeing an invalid object name in the input, pretend as if the bad input was not given.

--bisect

Pretend as if the bad bisection ref `refs/bisect/bad` was listed and as if it was followed by `--not` and the good bisection refs `refs/bisect/good-*` on the command line. Cannot be combined with --first-parent.

--stdin

In addition to the <*commit*> listed on the command line, read them from the standard input. If a -- separator is seen, stop reading commits and start reading paths to limit the result.

--cherry-mark

Like `--cherry-pick` (see below) but mark equivalent commits with `=` rather than omitting them, and inequivalent ones with `+`.

--cherry-pick

Omit any commit that introduces the same change as another commit on the “other side” when the set of commits are limited with symmetric difference.

For example, if you have two branches, `A` and `B`, a usual way to list all commits on only one side of them is with `--left-right` (see the example below in the description of the `--left-right` option). However, it shows the commits that were cherry-picked from the other branch (for example, “3rd on b” may be cherry-picked from branch A). With this option, such pairs of commits are excluded from the output.

--left-only

--right-only

List only commits on the respective side of a symmetric range, i.e. only those which would be marked `<` resp. `>` by `--left-right`.

For example, `--cherry-pick --right-only A...B` omits those commits from `B` which are in `A` or are patch-equivalent to a commit in `A`. In other words, this lists the `+` commits from `git cherry A B`. More precisely, `--cherry-pick --right-only --no-merges` gives the exact list.

--cherry

A synonym for `--right-only --cherry-mark --no-merges`; useful to limit the output to the commits on our side and mark those that have been applied to the other side of a forked history with `git log --cherry upstream...mybranch`, similar to `git cherry upstream mybranch`.

-g

--walk-reflogs

Instead of walking the commit ancestry chain, walk reflog entries from the most recent one to older ones. When this option is used you cannot specify commits to exclude (that is, *^commit*, *commit1..commit2*, and *commit1...commit2* notations cannot be used).

With `--pretty` format other than `oneline` (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. By default, *commit@{Nth}* notation is used in the output. When the starting commit is specified as *commit@{now}*, output also uses *commit@{timestamp}* notation instead. Under `--pretty=oneline`, the commit message is prefixed with this information on the same line. This option cannot be combined with `--reverse`. See also git-reflog[1].

--merge

After a failed merge, show refs that touch files having a conflict and don't exist on all heads to merge.

--boundary

Output excluded boundary commits. Boundary commits are prefixed with `-`.

History Simplification

Sometimes you are only interested in parts of the history, for example the commits modifying a particular <path>. But there are two parts of *History Simplification*, one part is selecting the commits and the other is how to do it, as there are various strategies to simplify the history.

The following options select the commits to be shown:

<paths>

Commits modifying the given <paths> are selected.

--simplify-by-decoration

Commits that are referred by some branch or tag are selected.

Note that extra commits can be shown to give a meaningful history.

The following options affect the way the simplification is performed:

Default mode

Simplifies the history to the simplest history explaining the final state of the tree. Simplest because it prunes some side branches if the end result is the same (i.e. merging branches with the same content)

--full-history

Same as the default mode, but does not prune some history.

--dense

Only the selected commits are shown, plus some to have a meaningful history.

--sparse

All commits in the simplified history are shown.

--simplify-merges

Additional option to `--full-history` to remove some needless merges from the resulting history, as there are no selected commits contributing to this merge.

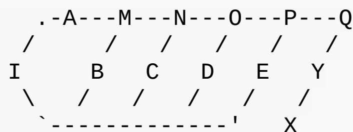
--ancestry-path

When given a range of commits to display (e.g. *commit1..commit2* or *commit2 ^commit1*), only display commits that exist directly on the ancestry chain between the *commit1* and *commit2*, i.e. commits that are both descendants of *commit1*, and ancestors of *commit2*.

A more detailed explanation follows.

Suppose you specified `foo` as the <paths>. We shall call commits that modify `foo` !TREESAME, and the rest TREESAME. (In a diff filtered for `foo`, they look different and equal, respectively.)

In the following, we will always refer to the same example history to illustrate the differences between simplification settings. We assume that you are filtering for a file `foo` in this commit graph:



```
     .-A---M---N---O---P---Q
    /     /   /   /   /   /
   I     B   C   D   E   Y
    \   /   /   /   /   /
     `-------------'   X
```

The horizontal line of history A---Q is taken to be the first parent of each merge. The commits are:

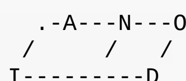
- `I` is the initial commit, in which `foo` exists with contents “asdf”, and a file `quux` exists with contents “quux”. Initial commits are compared to an empty tree, so `I` is !TREESAME.
- In `A` , `foo` contains just “foo”.
- `B` contains the same change as `A` . Its merge `M` is trivial and hence TREESAME to all parents.
- `C` does not change `foo` , but its merge `N` changes it to “foobar”, so it is not TREESAME to any parent.
- `D` sets `foo` to “baz”. Its merge `O` combines the strings from `N` and `D` to “foobarbaz”; i.e., it is not TREESAME to any parent.
- `E` changes `quux` to “xyzzy”, and its merge `P` combines the strings to “quux xyzzy”. `P` is TREESAME to `O` , but not to `E` .
- `X` is an independent root commit that added a new file `side` , and `Y` modified it. `Y` is TREESAME to `X` . Its merge `Q` added `side` to `P` , and `Q` is TREESAME to `P` , but not to `Y` .

`rev-list` walks backwards through history, including or excluding commits based on whether `--full-history` and/or parent rewriting (via `--parents` or `--children`) are used. The following settings are available.

Default mode

Commits are included if they are not TREESAME to any parent (though this can be changed, see `--sparse` below). If the commit was a merge, and it was TREESAME to one parent, follow only that parent. (Even if there are several TREESAME parents, follow only one of them.) Otherwise, follow all parents.

This results in:



```
     .-A---N---O
    /     /   /
   I---------D
```

Note how the rule to only follow the TREESAME parent, if one is available, removed `B` from consideration entirely. `C` was considered via `N`, but is TREESAME. Root commits are compared to an empty tree, so `I` is !TREESAME.

Parent/child relations are only visible with `--parents`, but that does not affect the commits selected in default mode, so we have shown the parent lines.

--full-history without parent rewriting

This mode differs from the default in one point: always follow all parents of a merge, even if it is TREESAME to one of them. Even if more than one side of the merge has commits that are included, this does not imply that the merge itself is! In the example, we get

```
        I  A  B  N  D  O  P  Q
```

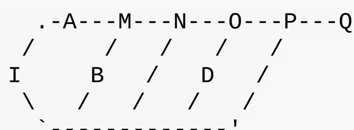
`M` was excluded because it is TREESAME to both parents. `E`, `C` and `B` were all walked, but only `B` was !TREESAME, so the others do not appear.

Note that without parent rewriting, it is not really possible to talk about the parent/child relationships between the commits, so we show them disconnected.

--full-history with parent rewriting

Ordinary commits are only included if they are !TREESAME (though this can be changed, see `--sparse` below).

Merges are always included. However, their parent list is rewritten: Along each parent, prune away commits that are not included themselves. This results in



```
          .-A---M---N---O---P---Q
         /     /   /   /   /
        I     B   /   D   /
         \   /   /   /   /
          `-------------'
```

Compare to `--full-history` without rewriting above. Note that `E` was pruned away because it is TREESAME, but the parent list of P was rewritten to contain `E`'s parent `I`. The same happened for `C` and `N`, and `X`, `Y` and `Q`.

In addition to the above settings, you can change whether TREESAME affects inclusion:

--dense

Commits that are walked are included if they are not TREESAME to any parent.

--sparse

All commits that are walked are included.

Note that without `--full-history`, this still simplifies merges: if one of the parents is TREESAME, we follow only that one, so the other sides of the merge are never walked.

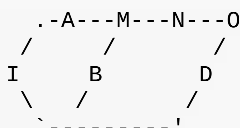
--simplify-merges

First, build a history graph in the same way that `--full-history` with parent rewriting does (see above).

Then simplify each commit `C` to its replacement `C'` in the final history according to the following rules:

- Set `C'` to `C`.
- Replace each parent `P` of `C'` with its simplification `P'`. In the process, drop parents that are ancestors of other parents or that are root commits TREESAME to an empty tree, and remove duplicates, but take care to never drop all parents that we are TREESAME to.
- If after this parent rewriting, `C'` is a root or merge commit (has zero or >1 parents), a boundary commit, or !TREESAME, it remains. Otherwise, it is replaced with its only parent.

The effect of this is best shown by way of comparing to `--full-history` with parent rewriting. The example turns into:



```
  .-A---M---N---O
 /     /       /
I     B       D
 \   /       /
  `---------'
```

Note the major differences in `N`, `P`, and `Q` over `--full-history`:

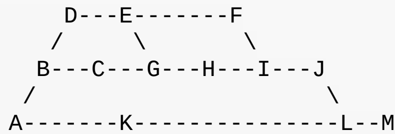
- `N`'s parent list had `I` removed, because it is an ancestor of the other parent `M`. Still, `N` remained because it is !TREESAME.
- `P`'s parent list similarly had `I` removed. `P` was then removed completely, because it had one parent and is TREESAME.
- `Q`'s parent list had `Y` simplified to `X`. `X` was then removed, because it was a TREESAME root. `Q` was then removed completely, because it had one parent and is TREESAME.

Finally, there is a fifth simplification mode available:

--ancestry-path

Limit the displayed commits to those directly on the ancestry chain between the “from” and “to” commits in the given commit range. I.e. only display commits that are ancestor of the “to” commit and descendants of the “from” commit.

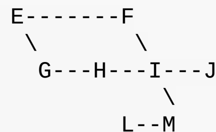
As an example use case, consider the following commit history:



```
            D---E-------F
           /     \       \
          B---C---G---H---I---J
         /                     \
        A-------K---------------L--M
```

A regular *D..M* computes the set of commits that are ancestors of `M`, but excludes the ones that are ancestors of `D`. This is useful to see what happened to the history leading to `M` since `D`, in the sense that “what does `M` have that did not exist in `D`”. The result in this example would be all the commits, except `A` and `B` (and `D` itself, of course).

When we want to find out what commits in `M` are contaminated with the bug introduced by `D` and need fixing, however, we might want to view only the subset of *D..M* that are actually descendants of `D`, i.e. excluding `C` and `K`. This is exactly what the `--ancestry-path` option does. Applied to the *D..M* range, it results in:



```
            E-------F
             \       \
              G---H---I---J
                           \
                        L--M
```

The `--simplify-by-decoration` option allows you to view only the big picture of the topology of the history, by omitting commits that are not referenced by tags. Commits are marked as !TREESAME (in other words, kept after history simplification rules described above) if (1) they are referenced by tags, or (2) they change the contents of the paths given on the command line. All other commits are marked as TREESAME (subject to be simplified away).

Commit Ordering

By default, the commits are shown in reverse chronological order.

--date-order

Show no parents before all of its children are shown, but otherwise show commits in the commit timestamp order.

--author-date-order

Show no parents before all of its children are shown, but otherwise show commits in the author timestamp order.

--topo-order

Show no parents before all of its children are shown, and avoid showing commits on multiple lines of history intermixed.

For example, in a commit history like this:

```
---1----2----4----7
\            \
 3----5----6----8---
```

where the numbers denote the order of commit timestamps, `git rev-list` and friends with `--date-order` show the commits in the timestamp order: 8 7 6 5 4 3 2 1.

With `--topo-order`, they would show 8 6 5 3 7 4 2 1 (or 8 7 4 2 6 5 3 1); some older commits are shown before newer ones in order to avoid showing the commits from two parallel development track mixed together.

--reverse

Output the commits in reverse order. Cannot be combined with `--walk-reflogs`.

Object Traversal

These options are mostly targeted for packing of Git repositories.

--no-walk[=(sorted|unsorted)]

Only show the given commits, but do not traverse their ancestors. This has no effect if a range is specified. If the argument `unsorted` is given, the commits are shown in the order they were given on the command line. Otherwise (if `sorted` or no argument was given), the commits are shown in reverse chronological order by commit time. Cannot be combined with `--graph`.

--do-walk

Overrides a previous `--no-walk`.

Commit Formatting

pretty-options.txt

--relative-date

Synonym for `--date=relative`.

--date=<format>

Only takes effect for dates shown in human-readable format, such as when using `--pretty`. `log.date` config variable sets a default value for the log command's `--date` option. By default, dates are shown in the original time zone (either committer's or author's). If `-local` is appended to the format (e.g., `iso-local`), the user's local time zone is used instead.

`--date=relative` shows dates relative to the current time, e.g. "2 hours ago". The `-local` option cannot be used with `--raw` or `--relative`.

`--date=local` is an alias for `--date=default-local`.

`--date=iso` (or `--date=iso8601`) shows timestamps in a ISO 8601-like format. The differences to the strict ISO 8601 format are:

- a space instead of the `T` date/time delimiter
- a space between time and time zone
- no colon between hours and minutes of the time zone

`--date=iso-strict` (or `--date=iso8601-strict`) shows timestamps in strict ISO 8601 format.

- `--date=rfc` (or `--date=rfc2822`) shows timestamps in RFC 2822 format, often found in email messages.
- `--date=short` shows only the date, but not the time, in `YYYY-MM-DD` format.
- `--date=raw` shows the date in the internal raw Git format `%s %z` format.
- `--date=format:...` feeds the format `...` to your system `strftime`. Use `--date=format:%c` to show the date in your system locale's preferred format. See the `strftime` manual for a complete list of format placeholders. When using `-local`, the correct syntax is `--date=format-local:...`.
- `--date=default` is the default format, and is similar to `--date=rfc2822`, with a few exceptions:
- there is no comma after the day-of-week
- the time zone is omitted when the local time zone is used

--parents

Print also the parents of the commit (in the form "commit parent…"). Also enables parent rewriting, see *History Simplification* below.

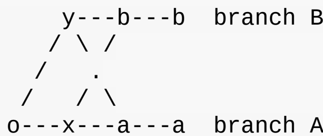
--children

Print also the children of the commit (in the form "commit child…"). Also enables parent rewriting, see *History Simplification* below.

--left-right

Mark which side of a symmetric diff a commit is reachable from. Commits from the left side are prefixed with `<` and those from the right with `>`. If combined with `--boundary`, those commits are prefixed with `-`.

For example, if you have this topology:



```
        y---b---b  branch B
       / \ /
      /   .
     /   / \
    o---x---a---a  branch A
```

you would get an output like this:

```
    $ git rev-list --left-right --boundary --pretty=oneline A...B

    &gt;bbbbbbb... 3rd on b
    &gt;bbbbbbb... 2nd on b
    &lt;aaaaaaa... 3rd on a
    &lt;aaaaaaa... 2nd on a
    -yyyyyyy... 1st on b
    -xxxxxxx... 1st on a
```

--graph

Draw a text-based graphical representation of the commit history on the left hand side of the output. This may cause extra lines to be printed in between commits, in order for the graph history to be drawn properly. Cannot be combined with `--no-walk`.

This enables parent rewriting, see *History Simplification* below.

This implies the `--topo-order` option by default, but the `--date-order` option may also be specified.

--show-linear-break[=<barrier>]

When --graph is not used, all history branches are flattened which can make it hard to see that the two consecutive commits do not belong to a linear branch. This option puts a barrier in between them in that case. If `<barrier>` is specified, it is the string that will be shown instead of the default one.

Diff Formatting

Listed below are options that control the formatting of diff output. Some of them are specific to git-rev-list[1], however other diff options may be given. See git-diff-files[1] for more options.

-c

With this option, diff output for a merge commit shows the differences from each of the parents to the merge result simultaneously instead of showing pairwise diff between a parent and the result one at a time. Furthermore, it lists only files which were modified from all parents.

--cc

This flag implies the `-c` option and further compresses the patch output by omitting uninteresting hunks whose contents in the parents have only two variants and the merge result picks one of them without modification.

-m

This flag makes the merge commits show the full diff like regular commits; for each merge parent, a separate log entry and diff is generated. An exception is that only diff against the first parent is shown when `--first-parent` option is given; in that case, the output represents the changes the merge brought *into* the then-current branch.

-r

Show recursive diffs.

-t

Show the tree objects in the diff output. This implies `-r`.

PRETTY FORMATS

If the commit is a merge, and if the pretty-format is not *oneline*, *email* or *raw*, an additional line is inserted before the *Author:* line. This line begins with "Merge: " and the sha1s of ancestral commits are printed, separated by spaces. Note that the listed commits may not necessarily be the list of the **direct** parent commits if you have limited your view of history: for example, if you are only interested in changes related to a certain directory or file.

There are several built-in formats, and you can define additional formats by setting a pretty.<name> config option to either another format name, or a *format:* string, as described below (see git-config[1]). Here are the details of the built-in formats:

- *oneline*

  ```
  &lt;sha1&gt; &lt;title line&gt;
  ```

  This is designed to be as compact as possible.

- *short*

  ```
  commit &lt;sha1&gt;
  Author: &lt;author&gt;
  ```

  ```
  &lt;title line&gt;
  ```

- *medium*

  ```
  commit &lt;sha1&gt;
  Author: &lt;author&gt;
  Date:   &lt;author date&gt;
  ```

  ```
  &lt;title line&gt;
  ```

  ```
  &lt;full commit message&gt;
  ```

- *full*

  ```
  commit &lt;sha1&gt;
  Author: &lt;author&gt;
  Commit: &lt;committer&gt;
  ```

  ```
  &lt;title line&gt;
  ```

  ```
  &lt;full commit message&gt;
  ```

- *fuller*

  ```
  commit &lt;sha1&gt;
  Author:     &lt;author&gt;
  AuthorDate: &lt;author date&gt;
  Commit:     &lt;committer&gt;
  CommitDate: &lt;committer date&gt;
  ```

  ```
  &lt;title line&gt;
  ```

  ```
  &lt;full commit message&gt;
  ```

- *email*

```
From &lt;sha1&gt; &lt;date&gt;
From: &lt;author&gt;
Date: &lt;author date&gt;
Subject: [PATCH] &lt;title line&gt;
```

```
&lt;full commit message&gt;
```

- *raw*

  The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the SHA-1s are displayed in full, regardless of whether --abbrev or --no-abbrev are used, and *parents* information show the true parent commits, without taking grafts or history simplification into account. Note that this format affects the way commits are displayed, but not the way the diff is shown e.g. with `git log --raw`. To get full object names in a raw diff format, use `--no-abbrev`.

- *format:<string>*

  The *format:<string>* format allows you to specify which information you want to show. It works a little bit like printf format, with the notable exception that you get a newline with *%n* instead of *\n*.

  E.g, *format:"The author of %h was %an, %ar%nThe title was >>%s<<%n"* would show something like this:

```
The author of fe6e0ee was Junio C Hamano, 23 hours ago
The title was &gt;&gt;t4119: test autocomputing -p&lt;n&gt; for traditional diff inpu
```

  The placeholders are:

  - *%H*: commit hash
  - *%h*: abbreviated commit hash
  - *%T*: tree hash
  - *%t*: abbreviated tree hash
  - *%P*: parent hashes
  - *%p*: abbreviated parent hashes
  - *%an*: author name
  - *%aN*: author name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
  - *%ae*: author email

- *%aE*: author email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ad*: author date (format respects --date= option)
- *%aD*: author date, RFC2822 style
- *%ar*: author date, relative
- *%at*: author date, UNIX timestamp
- *%ai*: author date, ISO 8601-like format
- *%aI*: author date, strict ISO 8601 format
- *%cn*: committer name
- *%cN*: committer name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ce*: committer email
- *%cE*: committer email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%cd*: committer date (format respects --date= option)
- *%cD*: committer date, RFC2822 style
- *%cr*: committer date, relative
- *%ct*: committer date, UNIX timestamp
- *%ci*: committer date, ISO 8601-like format
- *%cI*: committer date, strict ISO 8601 format
- *%d*: ref names, like the --decorate option of git-log[1]
- *%D*: ref names without the " (", ")" wrapping.
- *%e*: encoding
- *%s*: subject
- *%f*: sanitized subject line, suitable for a filename
- *%b*: body
- *%B*: raw body (unwrapped subject and body)
- *%N*: commit notes
- *%GG*: raw verification message from GPG for a signed commit

- *%G?*: show "G" for a Good signature, "B" for a Bad signature, "U" for a good, untrusted signature and "N" for no signature
- *%GS*: show the name of the signer for a signed commit
- *%GK*: show the key used to sign a signed commit
- *%gD*: reflog selector, e.g., `refs/stash@{1}`
- *%gd*: shortened reflog selector, e.g., `stash@{1}`
- *%gn*: reflog identity name
- *%gN*: reflog identity name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ge*: reflog identity email
- *%gE*: reflog identity email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%gs*: reflog subject
- *%Cred*: switch color to red
- *%Cgreen*: switch color to green
- *%Cblue*: switch color to blue
- *%Creset*: reset color
- *%C(…)*: color specification, as described in color.branch.* config option; adding `auto,` at the beginning will emit color only when colors are enabled for log output (by `color.diff`, `color.ui`, or `--color`, and respecting the `auto` settings of the former if we are going to a terminal). `auto` alone (i.e. `%C(auto)`) will turn on auto coloring on the next placeholders until the color is switched again.
- *%m*: left, right or boundary mark
- *%n*: newline
- *%%*: a raw %
- *%x00*: print a byte from a hex code
- *%w([<w>[,<i1>[,<i2>]]])*: switch line wrapping, like the -w option of git-shortlog[1].
- *%<(<N>[,trunc|ltrunc|mtrunc])*: make the next placeholder take at least N columns, padding spaces on the right if necessary. Optionally truncate at the beginning (ltrunc), the middle (mtrunc) or the end (trunc) if the output is longer than N columns. Note that truncating only works correctly with N >= 2.

  - *%<|(<N>)*: make the next placeholder take at least until Nth columns, padding spaces on the right if necessary
  - *%>(<N>)*, *%>|(<N>)*: similar to *%<(<N>)*, *%<|(<N>)* respectively, but padding spaces on the left
  - *%>>(<N>)*, *%>>|(<N>)*: similar to *%>(<N>)*, *%>|(<N>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces
  - *%><(<N>)*, *%><|(<N>)*: similar to *% <(<N>)*, *%<|(<N>)* respectively, but padding both sides (i.e. the text is centered)

> Note
>
> Some placeholders may depend on other options given to the revision traversal engine. For example, the `%g*` reflog options will insert an empty string unless we are traversing reflog entries (e.g., by `git log -g`). The `%d` and `%D` placeholders will use the "short" decoration format if `--decorate` was not already provided on the command line.

If you add a `+` (plus sign) after *%* of a placeholder, a line-feed is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

If you add a `-` (minus sign) after *%* of a placeholder, line-feeds that immediately precede the expansion are deleted if and only if the placeholder expands to an empty string.

If you add a ` ` (space) after *%* of a placeholder, a space is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

- *tformat:*

  The *tformat:* format works exactly like *format:*, except that it provides "terminator" semantics instead of "separator" semantics. In other words, each commit has the message terminator character (usually a newline) appended, rather than a separator placed between entries. This means that the final entry of a single-line format will be properly terminated with a new line, just as the "oneline" format does. For example:

  ```
  $ git log -2 --pretty=format:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973 -- NO NEWLINE

  $ git log -2 --pretty=tformat:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973
  ```

In addition, any unrecognized string that has a `%` in it is interpreted as if it has `tformat:` in front of it. For example, these two are equivalent:

```
$ git log -2 --pretty=tformat:%h 4da45bef
$ git log -2 --pretty=%h 4da45bef
```

COMMON DIFF OPTIONS

-p

-u

--patch

Generate patch (see section on generating patches).

-s

--no-patch

Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw

For each commit, show a summary of changes using the raw diff format. See the "RAW OUTPUT FORMAT" section of git-diff[1]. This is different from showing the log itself in raw format, which you can achieve with `--format=raw`.

--patch-with-raw

Synonym for `-p --raw`.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

`default` , `myers`

The basic greedy diff algorithm. Currently, this is the default.

`minimal`

Spend extra time to make sure the smallest possible diff is produced.

`patience`

Use "patience diff" algorithm when generating patches.

`histogram`

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured diff.algorithm variable to a non-default value and want to use the default one, then you have to use `--diff-algorithm=default` option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by `<width>`. The width of the filename part can be limited by giving another width `<name-width>` after a comma. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (does not affect `git format-patch`). By giving a third parameter `<count>`, you can limit the output to the first `<count>` lines, followed by `...` if there are more.

These parameters can also be set individually with `--stat-width=<width>`, `--stat-name-width=<name-width>` and `--stat-count=<count>`.

--numstat

Similar to `--stat` , but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two `-` instead of saying `0 0`.

--shortstat

Output only the last line of the `--stat` format containing total number of modified files, as well as number of added and deleted lines.

--dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of `--dirstat` can be customized by passing it a comma separated list of parameters. The defaults are controlled by the `diff.dirstat` configuration variable (see git-config[1]). The following parameters are available:

`changes`

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

`lines`

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive `--dirstat` behavior than the `changes` behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other `--*stat` options.

`files`

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest `--dirstat` behavior, since it does not have to look at the file contents at all.

`cumulative`

Count changes in a child directory for the parent directory as well. Note that when using `cumulative`, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the `noncumulative` parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: `--dirstat=files,10,cumulative`.

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for `-p --stat`.

-z

Separate the commits with NULs instead of with new newlines.

Also, when `--raw` or `--numstat` has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only

Show only names of changed files.

--name-status

Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]

Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule[1] `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--color[=<when>]

Show colored diff. `--color` (i.e. without =<*when*>) is the same as `--color=always`. <*when*> can be one of `always`, `never`, or `auto`.

--no-color

Turn off colored diff. It is the same as `--color=never`.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see `--word-diff-regex` below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies `--color`.

plain

Show words as `[-removed-]` and `{+added+}`. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption.
Added/removed/unchanged runs are printed in the usual unified diff format, starting with a `+` / `-` / `` ` character at the beginning of the line and extending to the end of the line. Newl``
~` on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies `--word-diff` unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append `|[^[:space:]]` to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, `--word-diff-regex=.` will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see gitattributes[1] or git-config[1]. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to `--word-diff=color` plus (if a regex was specified) `--word-diff-regex=<regex>`.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--check

Warn if changes introduce whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors on lines specified by <kind> in the color specified by `color.diff.whitespace`. <kind> is a comma separated list of `old`, `new`, `context`. When this option is not given, only whitespace errors in `new` lines are highlighted. E.g. `--ws-error-highlight=new,old` highlights whitespace errors on both deleted and added lines. `all` can be used as a short-hand for `old,new,context`.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to `--full-index`, output a binary diff that can be applied with `git-apply`.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. Non default number of digits can be specified with `--abbrev=<n>`.

-B[<n>][/<m>]

--break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything

new, and the number `m` controls this aspect of the -B option (defaults to 60%). `-B/70%` specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number `n` controls this aspect of the -B option (defaults to 50%). `-B20%` specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>]

--find-renames[=<n>]

If generating diffs, detect and report renames for each commit. For following files across renames while traversing history, see `--follow`. If `n` is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, `-M90%` means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a `%` sign, the number is to be read as a fraction, with a decimal point before it. I.e., `-M5` becomes 0.5, and is thus the same as `-M50%`. Similarly, `-M05` is the same as `-M5%`. To limit detection to exact renames, use `-M100%`. The default similarity index is 50%.

-C[<n>]

--find-copies[=<n>]

Detect copies as well as renames. See also `--find-copies-harder`. If `n` is specified, it has the same meaning as for `-M<n>`.

--find-copies-harder

For performance reasons, by default, `-C` option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one `-C` option has the same effect.

-D

--irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and `/dev/null`. The resulting patch is not meant to be applied with `patch` or `git apply`; this is solely for people who want to just concentrate on reviewing the text after

the change. In addition, the output obviously lack enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with `-B`, omit also the preimage in the deletion part of a delete/create pair.

-I<num>

The `-M` and `-C` options require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (`A`), Copied (`C`), Deleted (`D`), Modified (`M`), Renamed (`R`), have their type (i.e. regular file, symlink, submodule, …) changed (`T`), are Unmerged (`U`), are Unknown (`X`), or have had their pairing Broken (`B`). Any combination of the filter characters (including none) can be used. When `*` (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter’s use.

It is useful when you’re looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into `-S`, and keep going until you get the very first version of the block.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between `-S<regex> --pickaxe-regex` and `-G<regex>`, consider a commit with the following diff in the same file:

```
+    return !regexec(regexp, two-&gt;ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

See the *pickaxe* entry in gitdiffcore[7] for more information.

--pickaxe-all

When `-S` or `-G` finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to `-S` as an extended POSIX regular expression to match.

-O<orderfile>

Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config[1]). To cancel `diff.orderFile`, use `-O/dev/null`.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument.

-a

--text

Treat all files as text.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b

--ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w

--ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other.

-W

--function-context

Show whole surrounding functions of changes.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with gitattributes[5], you need to use this option with git-log[1] and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv

--no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See gitattributes[5] for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for git-diff[1] and git-log[1], but not for git-format-patch[1] or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in git-config[1] or gitmodules[5]. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

For more detailed explanation on these common options, see also gitdiffcore[7].

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The `a/` and `b/` filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, `/dev/null` is *not* used in place of the `a/` or `b/` filenames.

   When rename/copy is involved, `file1` and `file2` show the name of the source file of the rename/copy and the name of the file that rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

   ```
   old mode &lt;mode&gt;
   new mode &lt;mode&gt;
   deleted file mode &lt;mode&gt;
   new file mode &lt;mode&gt;
   copy from &lt;path&gt;
   copy to &lt;path&gt;
   rename from &lt;path&gt;
   rename to &lt;path&gt;
   similarity index &lt;number&gt;
   dissimilarity index &lt;number&gt;
   index &lt;hash&gt;..&lt;hash&gt; &lt;mode&gt;
   ```

   File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the `a/` and `b/` prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the SHA-1 checksum before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

4. All the `file1` files in the output refer to files before the commit, and all the `file2` files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

combined diff format

Any diff-generating command can take the `-c` or `--cc` option to produce a *combined diff* when showing a merge. This is the default format when showing merges with git-diff[1] or git-show[1]. Note also that you can give the `-m` option to any of these commands to force generation of diffs with individual parents of a merge.

A *combined diff* format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
    return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +    unsigned char sha1[20];
 +    struct commit *cmit;
    struct commit_list *list;
    static int initialized = 0;
    struct commit_name *n;

 +    if (get_sha1(arg, sha1) < 0)
 +        usage(describe_usage);
 +    cmit = lookup_commit_reference(sha1);
 +    if (!cmit)
 +        usage(describe_usage);
 +
    if (!initialized) {
        initialized = 1;
        for_each_ref(get_name);
```

1. It is preceded with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index &lt;hash&gt;;,&lt;hash&gt;;..&lt;hash&gt;;
   mode &lt;mode&gt;;,&lt;mode&gt;;..&lt;mode&gt;;
   new file mode &lt;mode&gt;;
   deleted file mode &lt;mode&gt;;,&lt;mode&gt;;
   ```

   The `mode <mode>;,<mode>;..<mode>;` line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.

4. Chunk header format is modified to prevent people from accidentally feeding it to `patch -p1`. Combined diff format was created for review of merge commit changes, and was not meant for apply. The change is similar to the change in the extended *index* header:

   ```
   @@@ &lt;from-file-range&gt; &lt;from-file-range&gt; &lt;to-file-range&gt; @@@
   ```

   There are (number of parents + 1) `@` characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has `-` (minus — appears in A but removed in B), `+` (plus — missing in A but added to B), or `" "` (space — unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X’s line is different from it.

A `-` character in the column N means that the line appears in fileN but it does not appear in the result. A `+` character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two `-` removals from both file1 and file2, plus `++` to mean one line that was added does not appear in either file1 or file2). Also eight other lines are the same from file1 but do not appear in file2 (hence prefixed with `+`).

When shown by `git diff-tree -c`, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by `git diff-files -c`, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

EXAMPLES

`git log --no-merges`

Show the whole commit history, but skip any merges

`git log v2.6.12.. include/scsi drivers/scsi`

Show all commits since version *v2.6.12* that changed any file in the `include/scsi` or `drivers/scsi` subdirectories

`git log --since="2 weeks ago" -- gitk`

Show the changes during the last two weeks to the file *gitk*. The “--” is necessary to avoid confusion with the **branch** named *gitk*

`git log --name-status release..test`

Show the commits that are in the "test" branch but not yet in the "release" branch, along with the list of paths each commit modifies.

`git log --follow builtin/rev-list.c`

Shows the commits that changed `builtin/rev-list.c`, including those commits that occurred before the file was given its present name.

`git log --branches --not --remotes=origin`

Shows all commits that are in any of local branches but not in any of remote-tracking branches for *origin* (what you have that origin doesn’t).

`git log master --not --remotes=*/master`

Shows all commits that are in local master but not in any remote repository master branches.

`git log -p -m --first-parent`

Shows the history including change diffs, but only from the “main branch” perspective, skipping commits that come from merged branches, and showing full diffs of changes introduced by the merges. This makes sense only when following a strict policy of merging all topic branches when staying on a single integration branch.

`git log -L '/int main/',/^}/:main.c`

Shows how the function `main()` in the file `main.c` evolved over time.

`git log -3`

Limits the number of commits to show to 3.

DISCUSSION

Git is to some extent character encoding agnostic.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.
- Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (`.git/config` (see git-config[1]), gitignore[5],

gitattributes[5] and gitmodules[5]).

Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that use legacy extended ASCII encodings. However, repositories created on such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.

- Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but *not* UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

Although we encourage that the commit log messages are encoded in UTF-8, both the core and Git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, Git does not forbid it. However, there are a few things to keep in mind.

1. *git commit* and *git commit-tree* issues a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have i18n.commitencoding in `.git/config` file, like this:

   ```
   [i18n]
       commitencoding = ISO-8859-1
   ```

   Commit objects created with the above setting record the value of `i18n.commitencoding` in its `encoding` header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.

2. *git log*, *git show*, *git blame* and friends look at the `encoding` header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with `i18n.logoutputencoding` in `.git/config` file, like this:

   ```
   [i18n]
       logoutputencoding = ISO-8859-1
   ```

   If you do not have this configuration variable, the value of `i18n.commitencoding` is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

CONFIGURATION

See git-config[1] for core variables and git-diff[1] for settings related to diff generation.

format.pretty

Default for the `--format` option. (See *Pretty Formats* above.) Defaults to `medium`.

i18n.logOutputEncoding

Encoding to use when displaying logs. (See *Discussion* above.) Defaults to the value of `i18n.commitEncoding` if set, and UTF-8 otherwise.

log.date

Default format for human-readable dates. (Compare the `--date` option.) Defaults to "default", which means to write dates like `Sat May 8 19:35:34 2010 -0500`.

log.follow

If `true`, `git log` will act as if the `--follow` option was used when a single <path> is given. This has the same limitations as `--follow`, i.e. it cannot be used to follow multiple files and does not work well on non-linear history.

log.showRoot

If `false`, `git log` and related commands will not treat the initial commit as a big creation event. Any root commits in `git log -p` output would be shown without a diff attached. The default is `true`.

mailmap.*

See git-shortlog[1].

notes.displayRef

Which refs, in addition to the default set by `core.notesRef` or *GIT_NOTES_REF*, to read notes from when showing commit messages with the `log` family of commands. See git-notes[1].

May be an unabbreviated ref name or a glob and may be specified multiple times. A warning will be issued for refs that do not exist, but a glob that does not match any refs is silently ignored.

This setting can be disabled by the `--no-notes` option, overridden by the *GIT_NOTES_DISPLAY_REF* environment variable, and overridden by the `--notes=<ref>` option.

GIT

Part of the git[1] suite

diff

NAME

git-diff - Show changes between commits, commit and working tree, etc

SYNOPSIS

```
git diff [options] [<commit>] [--] [<path>…]
git diff [options] --cached [<commit>] [--] [<path>…]
git diff [options] <commit> <commit> [--] [<path>…]
git diff [options] <blob> <blob>
git diff [options] [--no-index] [--] <path> <path>
```

DESCRIPTION

Show changes between the working tree and the index or a tree, changes between the index and a tree, changes between two trees, changes between two blob objects, or changes between two files on disk.

git diff [--options] [--] [<path>…]

This form is to view the changes you made relative to the index (staging area for the next commit). In other words, the differences are what you *could* tell Git to further add to the index but you still haven't. You can stage these changes by using git-add[1].

git diff --no-index [--options] [--] [<path>…]

This form is to compare the given two paths on the filesystem. You can omit the `--no-index` option when running the command in a working tree controlled by Git and at least one of the paths points outside the working tree, or when running the command outside a working tree controlled by Git.

git diff [--options] --cached [<commit>] [--] [<path>…]

This form is to view the changes you staged for the next commit relative to the named <commit>. Typically you would want comparison with the latest commit, so if you do not give <commit>, it defaults to HEAD. If HEAD does not exist (e.g. unborn branches) and <commit> is not given, it shows all staged changes. --staged is a synonym of --cached.

git diff [--options] <commit> [--] [<path>…]

This form is to view the changes you have in your working tree relative to the named <commit>. You can use HEAD to compare it with the latest commit, or a branch name to compare with the tip of a different branch.

git diff [--options] <commit> <commit> [--] [<path>…]

This is to view the changes between two arbitrary <commit>.

git diff [--options] <commit>..<commit> [--] [<path>…]

This is synonymous to the previous form. If <commit> on one side is omitted, it will have the same effect as using HEAD instead.

git diff [--options] <commit>...<commit> [--] [<path>…]

This form is to view the changes on the branch containing and up to the second <commit>, starting at a common ancestor of both <commit>. "git diff A...B" is equivalent to "git diff $(git-merge-base A B) B". You can omit any one of <commit>, which has the same effect as using HEAD instead.

Just in case if you are doing something exotic, it should be noted that all of the <commit> in the above description, except in the last two forms that use ".." notations, can be any <tree>.

For a more complete list of ways to spell <commit>, see "SPECIFYING REVISIONS" section in gitrevisions[7]. However, "diff" is about comparing two *endpoints*, not ranges, and the range notations ("<commit>..<commit>" and "<commit>...<commit>") do not mean a range as defined in the "SPECIFYING RANGES" section in gitrevisions[7].

git diff [options] <blob> <blob>

This form is to view the differences between the raw contents of two blob objects.

OPTIONS

-p

-u

--patch

Generate patch (see section on generating patches). This is the default.

-s

--no-patch

Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw

Generate the diff in raw format.

--patch-with-raw

Synonym for `-p --raw`.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

`default`, `myers`

The basic greedy diff algorithm. Currently, this is the default.

`minimal`

Spend extra time to make sure the smallest possible diff is produced.

`patience`

Use "patience diff" algorithm when generating patches.

`histogram`

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured diff.algorithm variable to a non-default value and want to use the default one, then you have to use `--diff-algorithm=default` option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by `<width>`. The width of the filename part can be limited by giving another width `<name-width>` after a comma. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (does not affect `git format-patch`). By giving a third parameter `<count>`, you can limit the output to the first `<count>` lines, followed by `...` if there are more.

These parameters can also be set individually with `--stat-width=<width>`, `--stat-name-width=<name-width>` and `--stat-count=<count>`.

--numstat

Similar to `--stat`, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two `-` instead of saying `0 0`.

--shortstat

Output only the last line of the `--stat` format containing total number of modified files, as well as number of added and deleted lines.

--dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of `--dirstat` can be customized by passing it a comma separated list of parameters. The defaults are controlled by the `diff.dirstat` configuration variable (see git-config[1]). The following parameters are available:

`changes`

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

`lines`

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive `--dirstat` behavior than the

`changes` behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other `--*stat` options.

`files`

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest `--dirstat` behavior, since it does not have to look at the file contents at all.

`cumulative`

Count changes in a child directory for the parent directory as well. Note that when using `cumulative`, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the `noncumulative` parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: `--dirstat=files,10,cumulative`.

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for `-p --stat`.

-z

When `--raw`, `--numstat`, `--name-only` or `--name-status` has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only

Show only names of changed files.

--name-status

Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]

Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule[1] `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--color[=<when>]

Show colored diff. `--color` (i.e. without =*<when>*) is the same as `--color=always`. *<when>* can be one of `always`, `never`, or `auto`. It can be changed by the `color.ui` and `color.diff` configuration settings.

--no-color

Turn off colored diff. This can be used to override configuration settings. It is the same as `--color=never`.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see `--word-diff-regex` below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies `--color`.

plain

Show words as `[-removed-]` and `{+added+}`. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a `+` / `-` / `` ` character at the beginning of the line and extending to the end of the line. Newl ``
~` on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies `--word-diff` unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append `|[^[:space:]]` to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, `--word-diff-regex=.` will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see gitattributes[1] or git-config[1]. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to `--word-diff=color` plus (if a regex was specified) `--word-diff-regex=<regex>`.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--check

Warn if changes introduce whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors on lines specified by <kind> in the color specified by `color.diff.whitespace`. <kind> is a comma separated list of `old`, `new`, `context`. When this option is not given, only whitespace errors in `new` lines are highlighted. E.g. `--ws-error-highlight=new,old` highlights whitespace errors on both deleted and added lines. `all` can be used as a short-hand for `old,new,context`.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to `--full-index`, output a binary diff that can be applied with `git-apply`.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. Non default number of digits can be specified with `--abbrev=<n>`.

-B[<n>][/<m>]

--break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number `m` controls this aspect of the -B option (defaults to 60%). `-B/70%` specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number `n` controls this aspect of the -B option (defaults to 50%). `-B20%` specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>]

--find-renames[=<n>]

Detect renames. If `n` is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, `-M90%` means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a `%` sign, the number is to be read as a fraction, with a decimal point before it. I.e., `-M5` becomes 0.5, and is thus the same as `-M50%`. Similarly, `-M05` is the same as `-M5%`. To limit detection to exact renames, use `-M100%`. The default similarity index is 50%.

-C[<n>]

--find-copies[=<n>]

Detect copies as well as renames. See also `--find-copies-harder`. If `n` is specified, it has the same meaning as for `-M<n>`.

--find-copies-harder

For performance reasons, by default, `-C` option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one `-C` option has the same effect.

-D

--irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and `/dev/null`. The resulting patch is not meant to be applied with `patch` or `git apply`; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lack enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with `-B`, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The `-M` and `-C` options require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (`A`), Copied (`C`), Deleted (`D`), Modified (`M`), Renamed (`R`), have their type (i.e. regular file, symlink, submodule, …) changed (`T`), are Unmerged (`U`), are Unknown (`X`), or have had their pairing Broken (`B`). Any combination of the filter characters (including none) can be used. When `*` (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter’s use.

It is useful when you’re looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into `-S`, and keep going until you get the very first version of the block.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between `-S<regex> --pickaxe-regex` and `-G<regex>`, consider a commit with the following diff in the same file:

```
+    return !regexec(regexp, two-&gt;ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

See the *pickaxe* entry in gitdiffcore[7] for more information.

--pickaxe-all

When `-S` or `-G` finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to `-S` as an extended POSIX regular expression to match.

-O<orderfile>

Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config[1]). To cancel `diff.orderFile`, use `-O/dev/null`.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument.

-a

--text

Treat all files as text.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b

--ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w

--ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other.

-W

--function-context

Show whole surrounding functions of changes.

--exit-code

Make the program exit with codes similar to diff(1). That is, it exits with 1 if there were differences and 0 means no differences.

--quiet

Disable all output of the program. Implies `--exit-code`.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with gitattributes[5], you need to use this option with git-log[1] and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv

--no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See gitattributes[5] for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for git-diff[1] and git-log[1], but not for git-format-patch[1] or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in git-config[1] or gitmodules[5]. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

For more detailed explanation on these common options, see also gitdiffcore[7].

<path>…

The <paths> parameters, when given, are used to limit the diff to the named paths (you can give directory names and get diff for all files under them).

Raw output format

The raw output format from "git-diff-index", "git-diff-tree", "git-diff-files" and "git diff --raw" are very similar.

These commands all compare two sets of things; what is compared differs:

git-diff-index <tree-ish>

compares the <tree-ish> and the files on the filesystem.

git-diff-index --cached <tree-ish>

compares the <tree-ish> and the index.

git-diff-tree [-r] <tree-ish-1> <tree-ish-2> [<pattern>…]

compares the trees named by the two arguments.

git-diff-files [<pattern>…]

compares the index and the files on the filesystem.

The "git-diff-tree" command begins its output by printing the hash of what is being compared. After that, all the commands print one output line per changed file.

An output line is formatted this way:

```
in-place edit  :100644 100644 bcd1234... 0123456... M file0
copy-edit      :100644 100644 abcd123... 1234567... C68 file1 file2
rename-edit    :100644 100644 abcd123... 1234567... R86 file1 file3
create         :000000 100644 0000000... 1234567... A file4
delete         :100644 000000 1234567... 0000000... D file5
unmerged       :000000 000000 0000000... 0000000... U file6
```

That is, from the left to the right:

1. a colon.
2. mode for "src"; 000000 if creation or unmerged.
3. a space.
4. mode for "dst"; 000000 if deletion or unmerged.
5. a space.
6. sha1 for "src"; 0{40} if creation or unmerged.
7. a space.
8. sha1 for "dst"; 0{40} if creation, unmerged or "look at work tree".

9. a space.
10. status, followed by optional "score" number.
11. a tab or a NUL when *-z* option is used.
12. path for "src"
13. a tab or a NUL when *-z* option is used; only exists for C or R.
14. path for "dst"; only exists for C or R.
15. an LF or a NUL when *-z* option is used, to terminate the record.

Possible status letters are:

- A: addition of a file
- C: copy of a file into a new one
- D: deletion of a file
- M: modification of the contents or mode of a file
- R: renaming of a file
- T: change in the type of the file
- U: file is unmerged (you must complete the merge before it can be committed)
- X: "unknown" change type (most probably a bug, please report it)

Status letters C and R are always followed by a score (denoting the percentage of similarity between the source and target of the move or copy). Status letter M may be followed by a score (denoting the percentage of dissimilarity) for file rewrites.

<sha1> is shown as all 0's if a file is new on the filesystem and it is out of sync with the index.

Example:

```
:100644 100644 5be4a4...... 000000...... M file.c
```

When `-z` option is not used, TAB, LF, and backslash characters in pathnames are represented as `\t`, `\n`, and `\\`, respectively.

diff format for merges

"git-diff-tree", "git-diff-files" and "git-diff --raw" can take *-c* or *--cc* option to generate diff output also for merge commits. The output differs from the format described above in the following way:

1. there is a colon for each parent
2. there are more "src" modes and "src" sha1
3. status is concatenated status characters for each parent
4. no optional "score" number
5. single path, only for "dst"

Example:

```
::100644 100644 100644 fabadb8... cc95eb0... 4866510... MM    describe.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The `a/` and `b/` filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, `/dev/null` is *not* used in place of the `a/` or `b/` filenames.

   When rename/copy is involved, `file1` and `file2` show the name of the source file of the rename/copy and the name of the file that rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

```
old mode &lt;mode&gt;
new mode &lt;mode&gt;
deleted file mode &lt;mode&gt;
new file mode &lt;mode&gt;
copy from &lt;path&gt;
copy to &lt;path&gt;
rename from &lt;path&gt;
rename to &lt;path&gt;
similarity index &lt;number&gt;
dissimilarity index &lt;number&gt;
index &lt;hash&gt;..&lt;hash&gt; &lt;mode&gt;
```

File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the `a/` and `b/` prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the SHA-1 checksum before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

4. All the `file1` files in the output refer to files before the commit, and all the `file2` files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

combined diff format

Any diff-generating command can take the `-c` or `--cc` option to produce a *combined diff* when showing a merge. This is the default format when showing merges with git-diff[1] or git-show[1]. Note also that you can give the `-m` option to any of these commands to force generation of diffs with individual parents of a merge.

A *combined diff* format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
    return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +    unsigned char sha1[20];
 +    struct commit *cmit;
    struct commit_list *list;
    static int initialized = 0;
    struct commit_name *n;

 +    if (get_sha1(arg, sha1) < 0)
 +        usage(describe_usage);
 +    cmit = lookup_commit_reference(sha1);
 +    if (!cmit)
 +        usage(describe_usage);
 +
    if (!initialized) {
        initialized = 1;
        for_each_ref(get_name);
```

1. It is preceded with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index &lt;hash&gt;;,&lt;hash&gt;;..&lt;hash&gt;;
   mode &lt;mode&gt;;,&lt;mode&gt;;..&lt;mode&gt;;
   new file mode &lt;mode&gt;;
   deleted file mode &lt;mode&gt;;,&lt;mode&gt;;
   ```

   The `mode <mode>;,<mode>;..<mode>;` line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.

4. Chunk header format is modified to prevent people from accidentally feeding it to `patch -p1`. Combined diff format was created for review of merge commit changes, and was not meant for apply. The change is similar to the change in the extended *index* header:

   ```
   @@@ &lt;from-file-range&gt; &lt;from-file-range&gt; &lt;to-file-range&gt; @@@
   ```

   There are (number of parents + 1) `@` characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has `-` (minus — appears in A but removed in B), `+` (plus — missing in A but added to B), or `" "` (space — unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X’s line is different from it.

A `-` character in the column N means that the line appears in fileN but it does not appear in the result. A `+` character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two `-` removals from both file1 and file2, plus `++` to mean one line that was added does not appear in either file1 or file2). Also eight other lines are the same from file1 but do not appear in file2 (hence prefixed with `+`).

When shown by `git diff-tree -c`, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by `git diff-files -c`, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

other diff formats

The `--summary` option describes newly added, deleted, renamed and copied files. The `--stat` option adds diffstat(1) graph to the output. These options can be combined with other options, such as `-p`, and are meant for human consumption.

When showing a change that involves a rename or a copy, `--stat` output formats the pathnames compactly by combining common prefix and suffix of the pathnames. For example, a change that moves `arch/i386/Makefile` to `arch/x86/Makefile` while modifying 4

lines will be shown like this:

```
arch/{i386 => x86}/Makefile    |   4 +--
```

The `--numstat` option gives the diffstat(1) information but is designed for easier machine consumption. An entry in `--numstat` output looks like this:

```
1    2    README
3    1    arch/{i386 => x86}/Makefile
```

That is, from left to right:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. pathname (possibly with rename/copy information);
6. a newline.

When `-z` output option is in effect, the output is formatted this way:

```
1    2    README NUL
3    1    NUL arch/i386/Makefile NUL arch/x86/Makefile NUL
```

That is:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. a NUL (only exists if renamed/copied);
6. pathname in preimage;
7. a NUL (only exists if renamed/copied);
8. pathname in postimage (only exists if renamed/copied);
9. a NUL.

The extra `NUL` before the preimage path in renamed case is to allow scripts that read the output to tell if the current record being read is a single-path record or a rename/copy record without reading ahead. After reading added and deleted lines, reading up to `NUL` would yield the pathname, but if that is `NUL`, the record will show two paths.

EXAMPLES

Various ways to check your working tree

```
$ git diff            (1)
$ git diff --cached   (2)
$ git diff HEAD       (3)
```

1. Changes in the working tree not yet staged for the next commit.
2. Changes between the index and your last commit; what you would be committing if you run "git commit" without "-a" option.
3. Changes in the working tree since your last commit; what you would be committing if you run "git commit -a"

Comparing with arbitrary commits

```
$ git diff test            (1)
$ git diff HEAD -- ./test  (2)
$ git diff HEAD^ HEAD      (3)
```

1. Instead of using the tip of the current branch, compare with the tip of "test" branch.
2. Instead of comparing with the tip of "test" branch, compare with the tip of the current branch, but limit the comparison to the file "test".
3. Compare the version before the last commit and the last commit.

Comparing branches

```
$ git diff topic master    (1)
$ git diff topic..master   (2)
$ git diff topic...master  (3)
```

1. Changes between the tips of the topic and the master branches.
2. Same as above.
3. Changes that occurred on the master branch since when the topic branch was started off it.

Limiting the diff output

```
$ git diff --diff-filter=MRC            (1)
$ git diff --name-status                (2)
$ git diff arch/i386 include/asm-i386   (3)
```

1. Show only modification, rename, and copy, but not addition or deletion.
2. Show only names and the nature of change, but not actual diff output.
3. Limit diff output to named subtrees.

Munging the diff output

```
$ git diff --find-copies-harder -B -C  (1)
$ git diff -R                          (2)
```

1. Spend extra cycles to find renames, copies and complete rewrites (very expensive).
2. Output diff in reverse.

SEE ALSO

diff(1), git-difftool[1], git-log[1], gitdiffcore[7], git-format-patch[1], git-apply[1]

GIT

Part of the git[1] suite

shortlog

NAME

git-shortlog - Summarize *git log* output

SYNOPSIS

```
git log --pretty=short | git shortlog [<options>]
git shortlog [<options>] [<revision range>] [[\--] <path>…]
```

DESCRIPTION

Summarizes *git log* output in a format suitable for inclusion in release announcements. Each commit will be grouped by author and title.

Additionally, "[PATCH]" will be stripped from the commit description.

If no revisions are passed on the command line and either standard input is not a terminal or there is no current branch, *git shortlog* will output a summary of the log read from standard input, without reference to the current repository.

OPTIONS

-n

--numbered

Sort output according to the number of commits per author instead of author alphabetic order.

-s

--summary

Suppress commit description and provide a commit count summary only.

-e

--email

Show the email address of each author.

--format[=<format>]

Instead of the commit subject, use some other information to describe each commit. *<format>* can be any string accepted by the `--format` option of *git log*, such as * *[%h] %s*. (See the "PRETTY FORMATS" section of git-log[1].)

```
Each pretty-printed commit will be rewrapped before it is shown.
```

-w[<width>[,<indent1>[,<indent2>]]]

Linewrap the output by wrapping each line at `width`. The first line of each entry is indented by `indent1` spaces, and the second and subsequent lines are indented by `indent2` spaces. `width`, `indent1`, and `indent2` default to 76, 6 and 9 respectively.

If width is `0` (zero) then indent the lines of the output without wrapping them.

<revision range>

Show only commits in the specified revision range. When no <revision range> is specified, it defaults to `HEAD` (i.e. the whole history leading to the current commit). `origin..HEAD` specifies all the commits reachable from the current commit (i.e. `HEAD`), but not from `origin`. For a complete list of ways to spell <revision range>, see the "Specifying Ranges" section of gitrevisions[7].

[--] <path>…

Consider only commits that are enough to explain how the files that match the specified paths came to be.

Paths may need to be prefixed with "-- " to separate them from options or the revision range, when confusion arises.

MAPPING AUTHORS

The `.mailmap` feature is used to coalesce together commits by the same person in the shortlog, where their name and/or email address was spelled differently.

If the file `.mailmap` exists at the toplevel of the repository, or at the location pointed to by the mailmap.file or mailmap.blob configuration options, it is used to map author and committer names and email addresses to canonical real names and email addresses.

In the simple form, each line in the file consists of the canonical real name of an author, whitespace, and an email address used in the commit (enclosed by < and >) to map to the name. For example:

```
Proper Name <commit@email.xx>
```

The more complex forms are:

```
<proper@email.xx> <commit@email.xx>
```

which allows mailmap to replace only the email part of a commit, and:

```
Proper Name <proper@email.xx> <commit@email.xx>
```

which allows mailmap to replace both the name and the email of a commit matching the specified commit email address, and:

```
Proper Name <proper@email.xx> Commit Name <commit@email.xx>
```

which allows mailmap to replace both the name and the email of a commit matching both the specified commit name and email address.

Example 1: Your history contains commits by two authors, Jane and Joe, whose names appear in the repository under several forms:

```
Joe Developer <joe@example.com>
Joe R. Developer <joe@example.com>
Jane Doe <jane@example.com>
Jane Doe <jane@laptop.(none)>
Jane D. <jane@desktop.(none)>
```

Now suppose that Joe wants his middle name initial used, and Jane prefers her family name fully spelled out. A proper `.mailmap` file would look like:

```
Jane Doe         <jane@desktop.(none)>
Joe R. Developer <joe@example.com>
```

Note how there is no need for an entry for `<jane@laptop.(none)>`, because the real name of that author is already correct.

Example 2: Your repository contains commits from the following authors:

```
nick1 <bugs@company.xx>
nick2 <bugs@company.xx>
nick2 <nick2@company.xx>
santa <me@company.xx>
claus <me@company.xx>
CTO <cto@coompany.xx>
```

Then you might want a `.mailmap` file that looks like:

```
<cto@company.xx>                       <cto@coompany.xx>
Some Dude <some@dude.xx>         nick1 <bugs@company.xx>
Other Author <other@author.xx>   nick2 <bugs@company.xx>
Other Author <other@author.xx>         <nick2@company.xx>
Santa Claus <santa.claus@northpole.xx> <me@company.xx>
```

Use hash # for comments that are either on their own line, or after the email address.

GIT

Part of the git[1] suite

describe

NAME

git-describe - Describe a commit using the most recent tag reachable from it

SYNOPSIS

```
git describe [--all] [--tags] [--contains] [--abbrev=<n>] [<commit-ish>…]
git describe [--all] [--tags] [--contains] [--abbrev=<n>] --dirty[=<mark>]
```

DESCRIPTION

The command finds the most recent tag that is reachable from a commit. If the tag points to the commit, then only the tag is shown. Otherwise, it suffixes the tag name with the number of additional commits on top of the tagged object and the abbreviated object name of the most recent commit.

By default (without --all or --tags) `git describe` only shows annotated tags. For more information about creating annotated tags see the -a and -s options to git-tag[1].

OPTIONS

<commit-ish>…

Commit-ish object names to describe. Defaults to HEAD if omitted.

--dirty[=<mark>]

Describe the working tree. It means describe HEAD and appends <mark> (`-dirty` by default) if the working tree is dirty.

--all

Instead of using only the annotated tags, use any ref found in `refs/` namespace. This option enables matching any known branch, remote-tracking branch, or lightweight tag.

--tags

Instead of using only the annotated tags, use any tag found in `refs/tags` namespace. This option enables matching a lightweight (non-annotated) tag.

--contains

Instead of finding the tag that predates the commit, find the tag that comes after the commit, and thus contains it. Automatically implies --tags.

--abbrev=<n>

Instead of using the default 7 hexadecimal digits as the abbreviated object name, use <n> digits, or as many digits as needed to form a unique object name. An <n> of 0 will suppress long format, only showing the closest tag.

--candidates=<n>

Instead of considering only the 10 most recent tags as candidates to describe the input commit-ish consider up to <n> candidates. Increasing <n> above 10 will take slightly longer but may produce a more accurate result. An <n> of 0 will cause only exact matches to be output.

--exact-match

Only output exact matches (a tag directly references the supplied commit). This is a synonym for --candidates=0.

--debug

Verbosely display information about the searching strategy being employed to standard error. The tag name will still be printed to standard out.

--long

Always output the long format (the tag, the number of commits and the abbreviated commit name) even when it matches a tag. This is useful when you want to see parts of the commit object name in "describe" output, even when the commit in question happens to be a tagged version. Instead of just emitting the tag name, it will describe such a commit as v1.2-0-gdeadbee (0th commit since tag v1.2 that points at object deadbee….).

--match <pattern>

Only consider tags matching the given `glob(7)` pattern, excluding the "refs/tags/" prefix. This can be used to avoid leaking private tags from the repository.

--always

Show uniquely abbreviated commit object as fallback.

--first-parent

Follow only the first parent commit upon seeing a merge commit. This is useful when you wish to not match tags on branches merged in the history of the target commit.

EXAMPLES

With something like git.git current tree, I get:

```
[torvalds@g5 git]$ git describe parent
v1.0.4-14-g2414721
```

i.e. the current head of my "parent" branch is based on v1.0.4, but since it has a few commits on top of that, describe has added the number of additional commits ("14") and an abbreviated object name for the commit itself ("2414721") at the end.

The number of additional commits is the number of commits which would be displayed by "git log v1.0.4..parent". The hash suffix is "-g" + 7-char abbreviation for the tip commit of parent (which was `2414721b194453f058079d897d13c4e377f92dc6`). The "g" prefix stands for "git" and is used to allow describing the version of a software depending on the SCM the software is managed with. This is useful in an environment where people may use different SCMs.

Doing a *git describe* on a tag-name will just show the tag name:

```
[torvalds@g5 git]$ git describe v1.0.4
v1.0.4
```

With --all, the command can use branch heads as references, so the output shows the reference path as well:

```
[torvalds@g5 git]$ git describe --all --abbrev=4 v1.0.5^2
tags/v1.0.0-21-g975b
```

```
[torvalds@g5 git]$ git describe --all --abbrev=4 HEAD^
heads/lt/describe-7-g975b
```

With --abbrev set to 0, the command can be used to find the closest tagname without any suffix:

```
[torvalds@g5 git]$ git describe --abbrev=0 v1.0.5^2
tags/v1.0.0
```

Note that the suffix you get if you type these commands today may be longer than what Linus saw above when he ran these commands, as your Git repository may have new commits whose object names begin with 975b that did not exist back then, and "-g975b" suffix alone may not be sufficient to disambiguate these commits.

SEARCH STRATEGY

For each commit-ish supplied, *git describe* will first look for a tag which tags exactly that commit. Annotated tags will always be preferred over lightweight tags, and tags with newer dates will always be preferred over tags with older dates. If an exact match is found, its name will be output and searching will stop.

If an exact match was not found, *git describe* will walk back through the commit history to locate an ancestor commit which has been tagged. The ancestor's tag will be output along with an abbreviation of the input commit-ish's SHA-1. If *--first-parent* was specified then the walk will only consider the first parent of each commit.

If multiple tags were found during the walk then the tag which has the fewest commits different from the input commit-ish will be selected and output. Here fewest commits different is defined as the number of commits which would be shown by `git log tag..input` will be the smallest number of commits possible.

GIT

Part of the git[1] suite

Patching

apply

NAME

git-apply - Apply a patch to files and/or to the index

SYNOPSIS

```
git apply [--stat] [--numstat] [--summary] [--check] [--index] [--3way]
          [--apply] [--no-add] [--build-fake-ancestor=<file>] [-R | --reverse]
          [--allow-binary-replacement | --binary] [--reject] [-z]
          [-p<n>] [-C<n>] [--inaccurate-eof] [--recount] [--cached]
          [--ignore-space-change | --ignore-whitespace ]
          [--whitespace=(nowarn|warn|fix|error|error-all)]
          [--exclude=<path>] [--include=<path>] [--directory=<root>]
          [--verbose] [--unsafe-paths] [<patch>…]
```

DESCRIPTION

Reads the supplied diff output (i.e. "a patch") and applies it to files. With the `--index` option the patch is also applied to the index, and with the `--cached` option the patch is only applied to the index. Without these options, the command applies the patch only to files, and does not require them to be in a Git repository.

This command applies the patch but does not create a commit. Use git-am[1] to create commits from patches generated by git-format-patch[1] and/or received by email.

OPTIONS

<patch>…

The files to read the patch from. - can be used to read from the standard input.

--stat

Instead of applying the patch, output diffstat for the input. Turns off "apply".

--numstat

Similar to `--stat`, but shows the number of added and deleted lines in decimal notation and the pathname without abbreviation, to make it more machine friendly. For binary files, outputs two `-` instead of saying `0 0`. Turns off "apply".

--summary

Instead of applying the patch, output a condensed summary of information obtained from git diff extended headers, such as creations, renames and mode changes. Turns off "apply".

--check

Instead of applying the patch, see if the patch is applicable to the current working tree and/or the index file and detects errors. Turns off "apply".

--index

When `--check` is in effect, or when applying the patch (which is the default when none of the options that disables it is in effect), make sure the patch is applicable to what the current index file records. If the file to be patched in the working tree is not up-to-date, it is flagged as an error. This flag also causes the index file to be updated.

--cached

Apply a patch without touching the working tree. Instead take the cached data, apply the patch, and store the result in the index without using the working tree. This implies `--index`.

-3

--3way

When the patch does not apply cleanly, fall back on 3-way merge if the patch records the identity of blobs it is supposed to apply to, and we have those blobs available locally, possibly leaving the conflict markers in the files in the working tree for the user to resolve. This option implies the `--index` option, and is incompatible with the `--reject` and the `--cached` options.

--build-fake-ancestor=<file>

Newer *git diff* output has embedded *index information* for each blob to help identify the original version that the patch applies to. When this flag is given, and if the original versions of the blobs are available locally, builds a temporary index containing those blobs.

When a pure mode change is encountered (which has no index information), the information is read from the current index instead.

-R

--reverse

Apply the patch in reverse.

--reject

For atomicity, *git apply* by default fails the whole patch and does not touch the working tree when some of the hunks do not apply. This option makes it apply the parts of the patch that are applicable, and leave the rejected hunks in corresponding *.rej files.

-z

When `--numstat` has been given, do not munge pathnames, but use a NUL-terminated machine-readable format.

Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

-p<n>

Remove <n> leading slashes from traditional diff paths. The default is 1.

-C<n>

Ensure at least <n> lines of surrounding context match before and after each change. When fewer lines of surrounding context exist they all must match. By default no context is ever ignored.

--unidiff-zero

By default, *git apply* expects that the patch being applied is a unified diff with at least one line of context. This provides good safety measures, but breaks down when applying a diff generated with `--unified=0`. To bypass these checks use `--unidiff-zero`.

Note, for the reasons stated above usage of context-free patches is discouraged.

--apply

If you use any of the options marked "Turns off *apply*" above, *git apply* reads and outputs the requested information without actually applying the patch. Give this flag after those flags to also apply the patch.

--no-add

When applying a patch, ignore additions made by the patch. This can be used to extract the common part between two files by first running *diff* on them and applying the result with this option, which would apply the deletion part but not the addition part.

--allow-binary-replacement

--binary

Historically we did not allow binary patch applied without an explicit permission from the user, and this flag was the way to do so. Currently we always allow binary patch application, so this is a no-op.

--exclude=<path-pattern>

Don’t apply changes to files matching the given path pattern. This can be useful when importing patchsets, where you want to exclude certain files or directories.

--include=<path-pattern>

Apply changes to files matching the given path pattern. This can be useful when importing patchsets, where you want to include certain files or directories.

When `--exclude` and `--include` patterns are used, they are examined in the order they appear on the command line, and the first match determines if a patch to each path is used. A patch to a path that does not match any include/exclude pattern is used by default if there is no include pattern on the command line, and ignored if there is any include pattern.

--ignore-space-change

--ignore-whitespace

When applying a patch, ignore changes in whitespace in context lines if necessary. Context lines will preserve their whitespace, and they will not undergo whitespace fixing regardless of the value of the `--whitespace` option. New lines will still be fixed, though.

--whitespace=<action>

When applying a patch, detect a new or modified line that has whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors.

By default, the command outputs warning messages but applies the patch. When `git-apply` is used for statistics and not applying a patch, it defaults to `nowarn`.

You can use different `<action>` values to control this behavior:

- `nowarn` turns off the trailing whitespace warning.
- `warn` outputs warnings for a few such errors, but applies the patch as-is (default).
- `fix` outputs warnings for a few such errors, and applies the patch after fixing them (`strip` is a synonym --- the tool used to consider only trailing whitespace characters as errors, and the fix involved *stripping* them, but modern Gits do more).

- `error` outputs warnings for a few such errors, and refuses to apply the patch.
- `error-all` is similar to `error` but shows all errors.

--inaccurate-eof

Under certain circumstances, some versions of *diff* do not correctly detect a missing new-line at the end of the file. As a result, patches created by such *diff* programs do not record incomplete lines correctly. This option adds support for applying such patches by working around this bug.

-v

--verbose

Report progress to stderr. By default, only a message about the current patch being applied will be printed. This option will cause additional information to be reported.

--recount

Do not trust the line counts in the hunk headers, but infer them by inspecting the patch (e.g. after editing the patch without adjusting the hunk headers appropriately).

--directory=<root>

Prepend <root> to all filenames. If a "-p" argument was also passed, it is applied before prepending the new root.

For example, a patch that talks about updating `a/git-gui.sh` to `b/git-gui.sh` can be applied to the file in the working tree `modules/git-gui/git-gui.sh` by running `git apply --directory=modules/git-gui`.

--unsafe-paths

By default, a patch that affects outside the working area (either a Git controlled working tree, or the current working directory when "git apply" is used as a replacement of GNU patch) is rejected as a mistake (or a mischief).

When `git apply` is used as a "better GNU patch", the user can pass the `--unsafe-paths` option to override this safety check. This option has no effect when `--index` or `--cached` is in use.

Configuration

apply.ignoreWhitespace

Set to *change* if you want changes in whitespace to be ignored by default. Set to one of: no, none, never, false if you want changes in whitespace to be significant.

apply.whitespace

When no `--whitespace` flag is given from the command line, this configuration item is used as the default.

Submodules

If the patch contains any changes to submodules then *git apply* treats these changes as follows.

If `--index` is specified (explicitly or implicitly), then the submodule commits must match the index exactly for the patch to apply. If any of the submodules are checked-out, then these check-outs are completely ignored, i.e., they are not required to be up-to-date or clean and they are not updated.

If `--index` is not specified, then the submodule commits in the patch are ignored and only the absence or presence of the corresponding subdirectory is checked and (if possible) updated.

SEE ALSO

git-am[1].

GIT

Part of the git[1] suite

cherry-pick

NAME

git-cherry-pick - Apply the changes introduced by some existing commits

SYNOPSIS

```
git cherry-pick [--edit] [-n] [-m parent-number] [-s] [-x] [--ff]
          [-S[<keyid>]] <commit>…
git cherry-pick --continue
git cherry-pick --quit
git cherry-pick --abort
```

DESCRIPTION

Given one or more existing commits, apply the change each one introduces, recording a new commit for each. This requires your working tree to be clean (no modifications from the HEAD commit).

When it is not obvious how to apply a change, the following happens:

1. The current branch and `HEAD` pointer stay at the last commit successfully made.
2. The `CHERRY_PICK_HEAD` ref is set to point at the commit that introduced the change that is difficult to apply.
3. Paths in which the change applied cleanly are updated both in the index file and in your working tree.
4. For conflicting paths, the index file records up to three versions, as described in the "TRUE MERGE" section of git-merge[1]. The working tree files will include a description of the conflict bracketed by the usual conflict markers <<<<<<< and >>>>>>> .
5. No other modifications are made.

See git-merge[1] for some hints on resolving such conflicts.

OPTIONS

<commit>…

Commits to cherry-pick. For a more complete list of ways to spell commits, see gitrevisions[7]. Sets of commits can be passed but no traversal is done by default, as if the *--no-walk* option was specified, see git-rev-list[1]. Note that specifying a range will feed all <commit>… arguments to a single revision walk (see a later example that uses *maint master..next*).

-e

--edit

With this option, *git cherry-pick* will let you edit the commit message prior to committing.

-x

When recording the commit, append a line that says "(cherry picked from commit …)" to the original commit message in order to indicate which commit this change was cherry-picked from. This is done only for cherry picks without conflicts. Do not use this option if you are cherry-picking from your private branch because the information is useless to the recipient. If on the other hand you are cherry-picking between two publicly visible branches (e.g. backporting a fix to a maintenance branch for an older release from a development branch), adding this information can be useful.

-r

It used to be that the command defaulted to do `-x` described above, and `-r` was to disable it. Now the default is not to do `-x` so this option is a no-op.

-m parent-number

--mainline parent-number

Usually you cannot cherry-pick a merge because you do not know which side of the merge should be considered the mainline. This option specifies the parent number (starting from 1) of the mainline and allows cherry-pick to replay the change relative to the specified parent.

-n

--no-commit

Usually the command automatically creates a sequence of commits. This flag applies the changes necessary to cherry-pick each named commit to your working tree and the index, without making any commit. In addition, when this option is used, your index does not have to match the HEAD commit. The cherry-pick is done against the beginning state of your index.

This is useful when cherry-picking more than one commits' effect to your index in a row.

-s

--signoff

Add Signed-off-by line at the end of the commit message. See the signoff option in git-commit[1] for more information.

-S[<keyid>]

--gpg-sign[=<keyid>]

GPG-sign commits. The `keyid` argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space.

--ff

If the current HEAD is the same as the parent of the cherry-pick'ed commit, then a fast forward to this commit will be performed.

--allow-empty

By default, cherry-picking an empty commit will fail, indicating that an explicit invocation of `git commit --allow-empty` is required. This option overrides that behavior, allowing empty commits to be preserved automatically in a cherry-pick. Note that when "--ff" is in effect, empty commits that meet the "fast-forward" requirement will be kept even without this option. Note also, that use of this option only keeps commits that were initially empty (i.e. the commit recorded the same tree as its parent). Commits which are made empty due to a previous commit are dropped. To force the inclusion of those commits use `--keep-redundant-commits`.

--allow-empty-message

By default, cherry-picking a commit with an empty message will fail. This option overrides that behaviour, allowing commits with empty messages to be cherry picked.

--keep-redundant-commits

If a commit being cherry picked duplicates a commit already in the current history, it will become empty. By default these redundant commits cause `cherry-pick` to stop so the user can examine the commit. This option overrides that behavior and creates an empty commit object. Implies `--allow-empty`.

--strategy=<strategy>

Use the given merge strategy. Should only be used once. See the MERGE STRATEGIES section in git-merge[1] for details.

-X<option>

--strategy-option=<option>

Pass the merge strategy-specific option through to the merge strategy. See git-merge[1] for details.

SEQUENCER SUBCOMMANDS

--continue

Continue the operation in progress using the information in *.git/sequencer*. Can be used to continue after resolving conflicts in a failed cherry-pick or revert.

--quit

Forget about the current operation in progress. Can be used to clear the sequencer state after a failed cherry-pick or revert.

--abort

Cancel the operation and return to the pre-sequence state.

EXAMPLES

`git cherry-pick master`

Apply the change introduced by the commit at the tip of the master branch and create a new commit with this change.

`git cherry-pick ..master`

`git cherry-pick ^HEAD master`

Apply the changes introduced by all commits that are ancestors of master but not of HEAD to produce new commits.

`git cherry-pick maint next ^master`

`git cherry-pick maint master..next`

Apply the changes introduced by all commits that are ancestors of maint or next, but not master or any of its ancestors. Note that the latter does not mean `maint` and everything between `master` and `next`; specifically, `maint` will not be used if it is included in `master`.

`git cherry-pick master~4 master~2`

Apply the changes introduced by the fifth and third last commits pointed to by master and create 2 new commits with these changes.

`git cherry-pick -n master~1 next`

Apply to the working tree and the index the changes introduced by the second last commit pointed to by master and by the last commit pointed to by next, but do not create any commit with these changes.

`git cherry-pick --ff ..next`

If history is linear and HEAD is an ancestor of next, update the working tree and advance the HEAD pointer to match next. Otherwise, apply the changes introduced by those commits that are in next but not HEAD to the current branch, creating a new commit for each new change.

`git rev-list --reverse master -- README | git cherry-pick -n --stdin`

Apply the changes introduced by all commits on the master branch that touched README to the working tree and index, so the result can be inspected and made into a single new commit if suitable.

The following sequence attempts to backport a patch, bails out because the code the patch applies to has changed too much, and then tries again, this time exercising more care about matching up context lines.

```
$ git cherry-pick topic^            (1)
$ git diff                          (2)
$ git reset --merge ORIG_HEAD       (3)
$ git cherry-pick -Xpatience topic^  (4)
```

1. apply the change that would be shown by `git show topic^`. In this example, the patch does not apply cleanly, so information about the conflict is written to the index and working tree and no new commit results.
2. summarize changes to be reconciled
3. cancel the cherry-pick. In other words, return to the pre-cherry-pick state, preserving any local modifications you had in the working tree.
4. try to apply the change introduced by `topic^` again, spending extra time to avoid mistakes based on incorrectly matching context lines.

SEE ALSO

git-revert[1]

GIT

Part of the git[1] suite

diff

NAME

git-diff - Show changes between commits, commit and working tree, etc

SYNOPSIS

```
git diff [options] [<commit>] [--] [<path>…]
git diff [options] --cached [<commit>] [--] [<path>…]
git diff [options] <commit> <commit> [--] [<path>…]
git diff [options] <blob> <blob>
git diff [options] [--no-index] [--] <path> <path>
```

DESCRIPTION

Show changes between the working tree and the index or a tree, changes between the index and a tree, changes between two trees, changes between two blob objects, or changes between two files on disk.

git diff [--options] [--] [<path>…]

This form is to view the changes you made relative to the index (staging area for the next commit). In other words, the differences are what you *could* tell Git to further add to the index but you still haven't. You can stage these changes by using git-add[1].

git diff --no-index [--options] [--] [<path>…]

This form is to compare the given two paths on the filesystem. You can omit the `--no-index` option when running the command in a working tree controlled by Git and at least one of the paths points outside the working tree, or when running the command outside a working tree controlled by Git.

git diff [--options] --cached [<commit>] [--] [<path>…]

This form is to view the changes you staged for the next commit relative to the named <commit>. Typically you would want comparison with the latest commit, so if you do not give <commit>, it defaults to HEAD. If HEAD does not exist (e.g. unborn branches) and <commit> is not given, it shows all staged changes. --staged is a synonym of --cached.

git diff [--options] <commit> [--] [<path>…]

This form is to view the changes you have in your working tree relative to the named <commit>. You can use HEAD to compare it with the latest commit, or a branch name to compare with the tip of a different branch.

git diff [--options] <commit> <commit> [--] [<path>…]

This is to view the changes between two arbitrary <commit>.

git diff [--options] <commit>..<commit> [--] [<path>…]

This is synonymous to the previous form. If <commit> on one side is omitted, it will have the same effect as using HEAD instead.

git diff [--options] <commit>...<commit> [--] [<path>…]

This form is to view the changes on the branch containing and up to the second <commit>, starting at a common ancestor of both <commit>. "git diff A...B" is equivalent to "git diff $(git-merge-base A B) B". You can omit any one of <commit>, which has the same effect as using HEAD instead.

Just in case if you are doing something exotic, it should be noted that all of the <commit> in the above description, except in the last two forms that use ".." notations, can be any <tree>.

For a more complete list of ways to spell <commit>, see "SPECIFYING REVISIONS" section in gitrevisions[7]. However, "diff" is about comparing two *endpoints*, not ranges, and the range notations ("<commit>..<commit>" and "<commit>...<commit>") do not mean a range as defined in the "SPECIFYING RANGES" section in gitrevisions[7].

git diff [options] <blob> <blob>

This form is to view the differences between the raw contents of two blob objects.

OPTIONS

-p

-u

--patch

Generate patch (see section on generating patches). This is the default.

-s

--no-patch

Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw

Generate the diff in raw format.

--patch-with-raw

Synonym for `-p --raw`.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

`default`, `myers`

The basic greedy diff algorithm. Currently, this is the default.

`minimal`

Spend extra time to make sure the smallest possible diff is produced.

`patience`

Use "patience diff" algorithm when generating patches.

`histogram`

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured diff.algorithm variable to a non-default value and want to use the default one, then you have to use `--diff-algorithm=default` option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by `<width>`. The width of the filename part can be limited by giving another width `<name-width>` after a comma. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (does not affect `git format-patch`). By giving a third parameter `<count>`, you can limit the output to the first `<count>` lines, followed by `...` if there are more.

These parameters can also be set individually with `--stat-width=<width>`, `--stat-name-width=<name-width>` and `--stat-count=<count>`.

--numstat

Similar to `--stat`, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two `-` instead of saying `0 0`.

--shortstat

Output only the last line of the `--stat` format containing total number of modified files, as well as number of added and deleted lines.

--dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of `--dirstat` can be customized by passing it a comma separated list of parameters. The defaults are controlled by the `diff.dirstat` configuration variable (see git-config[1]). The following parameters are available:

`changes`

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

`lines`

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive `--dirstat` behavior than the

`changes` behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other `--*stat` options.

`files`

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest `--dirstat` behavior, since it does not have to look at the file contents at all.

`cumulative`

Count changes in a child directory for the parent directory as well. Note that when using `cumulative`, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the `noncumulative` parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: `--dirstat=files,10,cumulative`.

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for `-p --stat`.

-z

When `--raw`, `--numstat`, `--name-only` or `--name-status` has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only

Show only names of changed files.

--name-status

Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]

Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule[1] `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--color[=<when>]

Show colored diff. `--color` (i.e. without =*<when>*) is the same as `--color=always`. *<when>* can be one of `always`, `never`, or `auto`. It can be changed by the `color.ui` and `color.diff` configuration settings.

--no-color

Turn off colored diff. This can be used to override configuration settings. It is the same as `--color=never`.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see `--word-diff-regex` below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies `--color`.

plain

Show words as `[-removed-]` and `{+added+}`. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a `+` / `-` / `` ` character at the beginning of the line and extending to the end of the line. Newl ``
~` on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies `--word-diff` unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append `|[^[:space:]]` to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, `--word-diff-regex=.` will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see gitattributes[1] or git-config[1]. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to `--word-diff=color` plus (if a regex was specified) `--word-diff-regex=<regex>`.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--check

Warn if changes introduce whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors on lines specified by <kind> in the color specified by `color.diff.whitespace`. <kind> is a comma separated list of `old`, `new`, `context`. When this option is not given, only whitespace errors in `new` lines are highlighted. E.g. `--ws-error-highlight=new,old` highlights whitespace errors on both deleted and added lines. `all` can be used as a short-hand for `old,new,context`.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to `--full-index`, output a binary diff that can be applied with `git-apply`.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. Non default number of digits can be specified with `--abbrev=<n>`.

-B[<n>][/<m>]

--break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number `m` controls this aspect of the -B option (defaults to 60%). `-B/70%` specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number `n` controls this aspect of the -B option (defaults to 50%). `-B20%` specifies that a change with addition and deletion compared to 20% or more of the file’s size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>]

--find-renames[=<n>]

Detect renames. If `n` is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file’s size). For example, `-M90%` means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn’t changed. Without a `%` sign, the number is to be read as a fraction, with a decimal point before it. I.e., `-M5` becomes 0.5, and is thus the same as `-M50%`. Similarly, `-M05` is the same as `-M5%`. To limit detection to exact renames, use `-M100%`. The default similarity index is 50%.

-C[<n>]

--find-copies[=<n>]

Detect copies as well as renames. See also `--find-copies-harder`. If `n` is specified, it has the same meaning as for `-M<n>`.

--find-copies-harder

For performance reasons, by default, `-C` option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one `-C` option has the same effect.

-D

--irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and `/dev/null`. The resulting patch is not meant to be applied with `patch` or `git apply`; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lack enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with `-B`, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The `-M` and `-C` options require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (`A`), Copied (`C`), Deleted (`D`), Modified (`M`), Renamed (`R`), have their type (i.e. regular file, symlink, submodule, …) changed (`T`), are Unmerged (`U`), are Unknown (`X`), or have had their pairing Broken (`B`). Any combination of the filter characters (including none) can be used. When `*` (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter’s use.

It is useful when you're looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into `-S`, and keep going until you get the very first version of the block.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between `-S<regex> --pickaxe-regex` and `-G<regex>`, consider a commit with the following diff in the same file:

```
+    return !regexec(regexp, two-&gt;ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

See the *pickaxe* entry in gitdiffcore[7] for more information.

--pickaxe-all

When `-S` or `-G` finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to `-S` as an extended POSIX regular expression to match.

-O<orderfile>

Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config[1]). To cancel `diff.orderFile`, use `-O/dev/null`.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument.

-a

--text

Treat all files as text.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b

--ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w

--ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other.

-W

--function-context

Show whole surrounding functions of changes.

--exit-code

Make the program exit with codes similar to diff(1). That is, it exits with 1 if there were differences and 0 means no differences.

--quiet

Disable all output of the program. Implies `--exit-code`.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with gitattributes[5], you need to use this option with git-log[1] and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv

--no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See gitattributes[5] for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for git-diff[1] and git-log[1], but not for git-format-patch[1] or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in git-config[1] or gitmodules[5]. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

For more detailed explanation on these common options, see also gitdiffcore[7].

<path>…

The <paths> parameters, when given, are used to limit the diff to the named paths (you can give directory names and get diff for all files under them).

Raw output format

The raw output format from "git-diff-index", "git-diff-tree", "git-diff-files" and "git diff --raw" are very similar.

These commands all compare two sets of things; what is compared differs:

git-diff-index <tree-ish>

compares the <tree-ish> and the files on the filesystem.

git-diff-index --cached <tree-ish>

compares the <tree-ish> and the index.

git-diff-tree [-r] <tree-ish-1> <tree-ish-2> [<pattern>…]

compares the trees named by the two arguments.

git-diff-files [<pattern>…]

compares the index and the files on the filesystem.

The "git-diff-tree" command begins its output by printing the hash of what is being compared. After that, all the commands print one output line per changed file.

An output line is formatted this way:

```
in-place edit  :100644 100644 bcd1234... 0123456... M file0
copy-edit      :100644 100644 abcd123... 1234567... C68 file1 file2
rename-edit    :100644 100644 abcd123... 1234567... R86 file1 file3
create         :000000 100644 0000000... 1234567... A file4
delete         :100644 000000 1234567... 0000000... D file5
unmerged       :000000 000000 0000000... 0000000... U file6
```

That is, from the left to the right:

1. a colon.
2. mode for "src"; 000000 if creation or unmerged.
3. a space.
4. mode for "dst"; 000000 if deletion or unmerged.
5. a space.
6. sha1 for "src"; 0{40} if creation or unmerged.
7. a space.
8. sha1 for "dst"; 0{40} if creation, unmerged or "look at work tree".

9. a space.
10. status, followed by optional "score" number.
11. a tab or a NUL when *-z* option is used.
12. path for "src"
13. a tab or a NUL when *-z* option is used; only exists for C or R.
14. path for "dst"; only exists for C or R.
15. an LF or a NUL when *-z* option is used, to terminate the record.

Possible status letters are:

- A: addition of a file
- C: copy of a file into a new one
- D: deletion of a file
- M: modification of the contents or mode of a file
- R: renaming of a file
- T: change in the type of the file
- U: file is unmerged (you must complete the merge before it can be committed)
- X: "unknown" change type (most probably a bug, please report it)

Status letters C and R are always followed by a score (denoting the percentage of similarity between the source and target of the move or copy). Status letter M may be followed by a score (denoting the percentage of dissimilarity) for file rewrites.

<sha1> is shown as all 0’s if a file is new on the filesystem and it is out of sync with the index.

Example:

```
:100644 100644 5be4a4...... 000000...... M file.c
```

When `-z` option is not used, TAB, LF, and backslash characters in pathnames are represented as `\t`, `\n`, and `\\`, respectively.

diff format for merges

"git-diff-tree", "git-diff-files" and "git-diff --raw" can take *-c* or *--cc* option to generate diff output also for merge commits. The output differs from the format described above in the following way:

1. there is a colon for each parent
2. there are more "src" modes and "src" sha1
3. status is concatenated status characters for each parent
4. no optional "score" number
5. single path, only for "dst"

Example:

```
::100644 100644 100644 fabadb8... cc95eb0... 4866510... MM    describe.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The `a/` and `b/` filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, `/dev/null` is *not* used in place of the `a/` or `b/` filenames.

   When rename/copy is involved, `file1` and `file2` show the name of the source file of the rename/copy and the name of the file that rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

```
old mode &lt;mode&gt;
new mode &lt;mode&gt;
deleted file mode &lt;mode&gt;
new file mode &lt;mode&gt;
copy from &lt;path&gt;
copy to &lt;path&gt;
rename from &lt;path&gt;
rename to &lt;path&gt;
similarity index &lt;number&gt;
dissimilarity index &lt;number&gt;
index &lt;hash&gt;..&lt;hash&gt; &lt;mode&gt;
```

File modes are printed as 6-digit octal numbers including the file type and file permission bits.

Path names in extended headers do not include the `a/` and `b/` prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the SHA-1 checksum before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

4. All the `file1` files in the output refer to files before the commit, and all the `file2` files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

combined diff format

Any diff-generating command can take the `-c` or `--cc` option to produce a *combined diff* when showing a merge. This is the default format when showing merges with git-diff[1] or git-show[1]. Note also that you can give the `-m` option to any of these commands to force generation of diffs with individual parents of a merge.

A *combined diff* format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
    return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +    unsigned char sha1[20];
 +    struct commit *cmit;
    struct commit_list *list;
    static int initialized = 0;
    struct commit_name *n;

 +    if (get_sha1(arg, sha1) < 0)
 +        usage(describe_usage);
 +    cmit = lookup_commit_reference(sha1);
 +    if (!cmit)
 +        usage(describe_usage);
 +
    if (!initialized) {
        initialized = 1;
        for_each_ref(get_name);
```

1. It is preceded with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index &lt;hash&gt;,&lt;hash&gt;..&lt;hash&gt;
   mode &lt;mode&gt;,&lt;mode&gt;..&lt;mode&gt;
   new file mode &lt;mode&gt;
   deleted file mode &lt;mode&gt;,&lt;mode&gt;
   ```

   The `mode <mode>,<mode>..<mode>` line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.

4. Chunk header format is modified to prevent people from accidentally feeding it to `patch -p1`. Combined diff format was created for review of merge commit changes, and was not meant for apply. The change is similar to the change in the extended *index* header:

   ```
   @@@ &lt;from-file-range&gt; &lt;from-file-range&gt; &lt;to-file-range&gt; @@@
   ```

   There are (number of parents + 1) `@` characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has `-` (minus — appears in A but removed in B), `+` (plus — missing in A but added to B), or `" "` (space — unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X’s line is different from it.

A `-` character in the column N means that the line appears in fileN but it does not appear in the result. A `+` character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two `-` removals from both file1 and file2, plus `++` to mean one line that was added does not appear in either file1 or file2). Also eight other lines are the same from file1 but do not appear in file2 (hence prefixed with `+`).

When shown by `git diff-tree -c`, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by `git diff-files -c`, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

other diff formats

The `--summary` option describes newly added, deleted, renamed and copied files. The `--stat` option adds diffstat(1) graph to the output. These options can be combined with other options, such as `-p`, and are meant for human consumption.

When showing a change that involves a rename or a copy, `--stat` output formats the pathnames compactly by combining common prefix and suffix of the pathnames. For example, a change that moves `arch/i386/Makefile` to `arch/x86/Makefile` while modifying 4

lines will be shown like this:

```
arch/{i386 => x86}/Makefile    |   4 +--
```

The `--numstat` option gives the diffstat(1) information but is designed for easier machine consumption. An entry in `--numstat` output looks like this:

```
1    2    README
3    1    arch/{i386 => x86}/Makefile
```

That is, from left to right:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. pathname (possibly with rename/copy information);
6. a newline.

When `-z` output option is in effect, the output is formatted this way:

```
1    2    README NUL
3    1    NUL arch/i386/Makefile NUL arch/x86/Makefile NUL
```

That is:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. a NUL (only exists if renamed/copied);
6. pathname in preimage;
7. a NUL (only exists if renamed/copied);
8. pathname in postimage (only exists if renamed/copied);
9. a NUL.

The extra `NUL` before the preimage path in renamed case is to allow scripts that read the output to tell if the current record being read is a single-path record or a rename/copy record without reading ahead. After reading added and deleted lines, reading up to `NUL` would yield the pathname, but if that is `NUL`, the record will show two paths.

EXAMPLES

Various ways to check your working tree

```
$ git diff            (1)
$ git diff --cached   (2)
$ git diff HEAD       (3)
```

1. Changes in the working tree not yet staged for the next commit.
2. Changes between the index and your last commit; what you would be committing if you run "git commit" without "-a" option.
3. Changes in the working tree since your last commit; what you would be committing if you run "git commit -a"

Comparing with arbitrary commits

```
$ git diff test            (1)
$ git diff HEAD -- ./test  (2)
$ git diff HEAD^ HEAD      (3)
```

1. Instead of using the tip of the current branch, compare with the tip of "test" branch.
2. Instead of comparing with the tip of "test" branch, compare with the tip of the current branch, but limit the comparison to the file "test".
3. Compare the version before the last commit and the last commit.

Comparing branches

```
$ git diff topic master    (1)
$ git diff topic..master   (2)
$ git diff topic...master  (3)
```

1. Changes between the tips of the topic and the master branches.
2. Same as above.
3. Changes that occurred on the master branch since when the topic branch was started off it.

Limiting the diff output

```
$ git diff --diff-filter=MRC            (1)
$ git diff --name-status                (2)
$ git diff arch/i386 include/asm-i386   (3)
```

1. Show only modification, rename, and copy, but not addition or deletion.
2. Show only names and the nature of change, but not actual diff output.
3. Limit diff output to named subtrees.

Munging the diff output

```
$ git diff --find-copies-harder -B -C  (1)
$ git diff -R                          (2)
```

1. Spend extra cycles to find renames, copies and complete rewrites (very expensive).
2. Output diff in reverse.

SEE ALSO

diff(1), git-difftool[1], git-log[1], gitdiffcore[7], git-format-patch[1], git-apply[1]

GIT

Part of the git[1] suite

rebase

NAME

git-rebase - Reapply commits on top of another base tip

SYNOPSIS

```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
	[<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
	--root [<branch>]
git rebase --continue | --skip | --abort | --edit-todo
```

DESCRIPTION

If <branch> is specified, *git rebase* will perform an automatic `git checkout <branch>` before doing anything else. Otherwise it remains on the current branch.

If <upstream> is not specified, the upstream configured in branch.<name>.remote and branch.<name>.merge options will be used (see git-config[1] for details) and the `--fork-point` option is assumed. If you are currently not on any branch or if the current branch does not have a configured upstream, the rebase will abort.

All changes made by commits in the current branch but that are not in <upstream> are saved to a temporary area. This is the same set of commits that would be shown by `git log <upstream>..HEAD` ; or by `git log 'fork_point'..HEAD` , if `--fork-point` is active (see the description on `--fork-point` below); or by `git log HEAD` , if the `--root` option is specified.

The current branch is reset to <upstream>, or <newbase> if the --onto option was supplied. This has the exact same effect as `git reset --hard <upstream>` (or <newbase>). ORIG_HEAD is set to point at the tip of the branch before the reset.

The commits that were previously saved into the temporary area are then reapplied to the current branch, one by one, in order. Note that any commits in HEAD which introduce the same textual changes as a commit in HEAD..<upstream> are omitted (i.e., a patch already accepted upstream with a different commit message or timestamp will be skipped).

It is possible that a merge failure will prevent this process from being completely automatic. You will have to resolve any such merge failure and run `git rebase --continue`. Another option is to bypass the commit that caused the merge failure with `git rebase --skip`. To check out the original <branch> and remove the .git/rebase-apply working files, use the command `git rebase --abort` instead.

Assume the following history exists and the current branch is "topic":

```
          A---B---C topic
         /
    D---E---F---G master
```

From this point, the result of either of the following commands:

```
git rebase master
git rebase master topic
```

would be:

```
                  A'--B'--C' topic
                 /
    D---E---F---G master
```

NOTE: The latter form is just a short-hand of `git checkout topic` followed by `git rebase master`. When rebase exits `topic` will remain the checked-out branch.

If the upstream branch already contains a change you have made (e.g., because you mailed a patch which was applied upstream), then that commit will be skipped. For example, running `git rebase master` on the following history (in which `A'` and `A` introduce the same set of changes, but have different committer information):

```
          A---B---C topic
         /
    D---E---A'---F master
```

will result in:

```
                   B'---C' topic
                  /
    D---E---A'---F master
```

Here is how you would transplant a topic branch based on one branch to another, to pretend that you forked the topic branch from the latter branch, using `rebase --onto`.

First let’s assume your *topic* is based on branch *next*. For example, a feature developed in *topic* depends on some functionality which is found in *next*.

```
      o---o---o---o---o  master
           \
            o---o---o---o---o  next
                             \
                              o---o---o  topic
```

We want to make *topic* forked from branch *master*; for example, because the functionality on which *topic* depends was merged into the more stable *master* branch. We want our tree to look like this:

```
      o---o---o---o---o  master
          |            \
          |             o'--o'--o'  topic
           \
            o---o---o---o---o  next
```

We can get this using the following command:

```
git rebase --onto master next topic
```

Another example of --onto option is to rebase part of a branch. If we have the following situation:

```
                              H---I---J topicB
                             /
                    E---F---G  topicA
                   /
      A---B---C---D  master
```

then the command

```
git rebase --onto master topicA topicB
```

would result in:

```
                   H'--I'--J'  topicB
                  /
                  | E---F---G  topicA
                  |/
      A---B---C---D  master
```

This is useful when topicB does not depend on topicA.

A range of commits could also be removed with rebase. If we have the following situation:

```
      E---F---G---H---I---J  topicA
```

then the command

```
git rebase --onto topicA~5 topicA~3 topicA
```

would result in the removal of commits F and G:

```
    E---H'---I'---J'  topicA
```

This is useful if F and G were flawed in some way, or should not be part of topicA. Note that the argument to --onto and the <upstream> parameter can be any valid commit-ish.

In case of conflict, *git rebase* will stop at the first problematic commit and leave conflict markers in the tree. You can use *git diff* to locate the markers (<<<<<<) and make edits to resolve the conflict. For each file you edit, you need to tell Git that the conflict has been resolved, typically this would be done with

```
git add <filename>
```

After resolving the conflict manually and updating the index with the desired resolution, you can continue the rebasing process with

```
git rebase --continue
```

Alternatively, you can undo the *git rebase* with

```
git rebase --abort
```

CONFIGURATION

rebase.stat

Whether to show a diffstat of what changed upstream since the last rebase. False by default.

rebase.autoSquash

If set to true enable *--autosquash* option by default.

rebase.autoStash

If set to true enable *--autostash* option by default.

rebase.missingCommitsCheck

If set to "warn", print warnings about removed commits in interactive mode. If set to "error", print the warnings and stop the rebase. If set to "ignore", no checking is done. "ignore" by default.

rebase.instructionFormat

Custom commit list format to use during an *--interactive* rebase.

OPTIONS

--onto <newbase>

Starting point at which to create the new commits. If the --onto option is not specified, the starting point is <upstream>. May be any valid commit, and not just an existing branch name.

As a special case, you may use "A...B" as a shortcut for the merge base of A and B if there is exactly one merge base. You can leave out at most one of A and B, in which case it defaults to HEAD.

<upstream>

Upstream branch to compare against. May be any valid commit, not just an existing branch name. Defaults to the configured upstream for the current branch.

<branch>

Working branch; defaults to HEAD.

--continue

Restart the rebasing process after having resolved a merge conflict.

--abort

Abort the rebase operation and reset HEAD to the original branch. If <branch> was provided when the rebase operation was started, then HEAD will be reset to <branch>. Otherwise HEAD will be reset to where it was when the rebase operation was started.

--keep-empty

Keep the commits that do not change anything from its parents in the result.

--skip

Restart the rebasing process by skipping the current patch.

--edit-todo

Edit the todo list during an interactive rebase.

-m

--merge

Use merging strategies to rebase. When the recursive (default) merge strategy is used, this allows rebase to be aware of renames on the upstream side.

Note that a rebase merge works by replaying each commit from the working branch on top of the <upstream> branch. Because of this, when a merge conflict happens, the side reported as *ours* is the so-far rebased series, starting with <upstream>, and *theirs* is the working branch. In other words, the sides are swapped.

-s <strategy>

--strategy=<strategy>

Use the given merge strategy. If there is no `-s` option *git merge-recursive* is used instead. This implies --merge.

Because *git rebase* replays each commit from the working branch on top of the <upstream> branch using the given strategy, using the *ours* strategy simply discards all patches from the <branch>, which makes little sense.

-X <strategy-option>

--strategy-option=<strategy-option>

Pass the <strategy-option> through to the merge strategy. This implies `--merge` and, if no strategy has been specified, `-s recursive`. Note the reversal of *ours* and *theirs* as noted above for the `-m` option.

-S[<keyid>]

--gpg-sign[=<keyid>]

GPG-sign commits. The `keyid` argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space.

-q

--quiet

Be quiet. Implies --no-stat.

-v

--verbose

Be verbose. Implies --stat.

--stat

Show a diffstat of what changed upstream since the last rebase. The diffstat is also controlled by the configuration option rebase.stat.

-n

--no-stat

Do not show a diffstat as part of the rebase process.

--no-verify

This option bypasses the pre-rebase hook. See also githooks[5].

--verify

Allows the pre-rebase hook to run, which is the default. This option can be used to override --no-verify. See also githooks[5].

-C<n>

Ensure at least <n> lines of surrounding context match before and after each change. When fewer lines of surrounding context exist they all must match. By default no context is ever ignored.

-f

--force-rebase

Force a rebase even if the current branch is up-to-date and the command without `--force` would return without doing anything.

You may find this (or --no-ff with an interactive rebase) helpful after reverting a topic branch merge, as this option recreates the topic branch with fresh commits so it can be remerged successfully without needing to "revert the reversion" (see the revert-a-faulty-merge How-To for details).

--fork-point

--no-fork-point

Use reflog to find a better common ancestor between <upstream> and <branch> when calculating which commits have been introduced by <branch>.

When --fork-point is active, *fork_point* will be used instead of <upstream> to calculate the set of commits to rebase, where *fork_point* is the result of `git merge-base --fork-point <upstream> <branch>` command (see git-merge-base[1]). If *fork_point* ends up being empty, the <upstream> will be used as a fallback.

If either <upstream> or --root is given on the command line, then the default is `--no-fork-point`, otherwise the default is `--fork-point`.

--ignore-whitespace

--whitespace=<option>

These flag are passed to the *git apply* program (see git-apply[1]) that applies the patch. Incompatible with the --interactive option.

--committer-date-is-author-date

--ignore-date

These flags are passed to *git am* to easily change the dates of the rebased commits (see git-am[1]). Incompatible with the --interactive option.

-i

--interactive

Make a list of the commits which are about to be rebased. Let the user edit that list before rebasing. This mode can also be used to split commits (see SPLITTING COMMITS below).

The commit list format can be changed by setting the configuration option rebase.instructionFormat. A customized instruction format will automatically have the long commit hash prepended to the format.

-p

--preserve-merges

Recreate merge commits instead of flattening the history by replaying commits a merge commit introduces. Merge conflict resolutions or manual amendments to merge commits are not preserved.

This uses the `--interactive` machinery internally, but combining it with the `--interactive` option explicitly is generally not a good idea unless you know what you are doing (see BUGS below).

-x <cmd>

--exec <cmd>

Append "exec <cmd>" after each line creating a commit in the final history. <cmd> will be interpreted as one or more shell commands.

This option can only be used with the `--interactive` option (see INTERACTIVE MODE below).

You may execute several commands by either using one instance of `--exec` with several commands:

```
git rebase -i --exec "cmd1 && cmd2 && ..."
```

or by giving more than one `--exec` :

```
git rebase -i --exec "cmd1" --exec "cmd2" --exec ...
```

If `--autosquash` is used, "exec" lines will not be appended for the intermediate commits, and will only appear at the end of each squash/fixup series.

--root

Rebase all commits reachable from <branch>, instead of limiting them with an <upstream>. This allows you to rebase the root commit(s) on a branch. When used with --onto, it will skip changes already contained in <newbase> (instead of <upstream>) whereas without --onto it will operate on every change. When used together with both --onto and --preserve-merges, *all* root commits will be rewritten to have <newbase> as parent instead.

--autosquash

--no-autosquash

When the commit log message begins with "squash! …" (or "fixup! …"), and there is a commit whose title begins with the same …, automatically modify the todo list of rebase -i so that the commit marked for squashing comes right after the commit to be modified, and change the action of the moved commit from `pick` to `squash` (or `fixup`). Ignores subsequent "fixup! " or "squash! " after the first, in case you referred to an earlier fixup/squash with `git commit --fixup/--squash` .

This option is only valid when the *--interactive* option is used.

If the *--autosquash* option is enabled by default using the configuration variable `rebase.autoSquash` , this option can be used to override and disable this setting.

--autostash

--no-autostash

Automatically create a temporary stash before the operation begins, and apply it after the operation ends. This means that you can run rebase on a dirty worktree. However, use with care: the final stash application after a successful rebase might result in non-trivial conflicts.

--no-ff

With --interactive, cherry-pick all rebased commits instead of fast-forwarding over the unchanged ones. This ensures that the entire history of the rebased branch is composed of new commits.

Without --interactive, this is a synonym for --force-rebase.

You may find this helpful after reverting a topic branch merge, as this option recreates the topic branch with fresh commits so it can be remerged successfully without needing to "revert the reversion" (see the revert-a-faulty-merge How-To for details).

MERGE STRATEGIES

The merge mechanism (`git merge` and `git pull` commands) allows the backend *merge strategies* to be chosen with `-s` option. Some strategies can also take their own options, which can be passed by giving `-X<option>` arguments to `git merge` and/or `git pull` .

resolve

This can only resolve two heads (i.e. the current branch and another branch you pulled from) using a 3-way merge algorithm. It tries to carefully detect criss-cross merge ambiguities and is considered generally safe and fast.

recursive

This can only resolve two heads using a 3-way merge algorithm. When there is more than one common ancestor that can be used for 3-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the 3-way merge. This has been reported to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from Linux 2.6 kernel development history. Additionally this can detect and handle merges involving renames. This is the default merge strategy when pulling or merging one branch.

The *recursive* strategy can take the following options:

ours

This option forces conflicting hunks to be auto-resolved cleanly by favoring *our* version. Changes from the other tree that do not conflict with our side are reflected to the merge result. For a binary file, the entire contents are taken from our side.

This should not be confused with the *ours* merge strategy, which does not even look at what the other tree contains at all. It discards everything the other tree did, declaring *our* history contains all that happened in it.

theirs

This is the opposite of *ours*.

patience

With this option, *merge-recursive* spends a little extra time to avoid mismerges that sometimes occur due to unimportant matching lines (e.g., braces from distinct functions). Use this when the branches to be merged have diverged wildly. See also git-diff[1] `--patience`.

diff-algorithm=[patience|minimal|histogram|myers]

Tells *merge-recursive* to use a different diff algorithm, which can help avoid mismerges that occur due to unimportant matching lines (such as braces from distinct functions). See also git-diff[1] `--diff-algorithm`.

ignore-space-change

ignore-all-space

ignore-space-at-eol

Treats lines with the indicated type of whitespace change as unchanged for the sake of a three-way merge. Whitespace changes mixed with other changes to a line are not ignored. See also git-diff[1] `-b`, `-w`, and `--ignore-space-at-eol`.

- If *their* version only introduces whitespace changes to a line, *our* version is used;
- If *our* version introduces whitespace changes but *their* version includes a substantial change, *their* version is used;
- Otherwise, the merge proceeds in the usual way.

renormalize

This runs a virtual check-out and check-in of all three stages of a file when resolving a three-way merge. This option is meant to be used when merging branches with different clean filters or end-of-line normalization rules. See "Merging branches with differing checkin/checkout attributes" in gitattributes[5] for details.

no-renormalize

Disables the `renormalize` option. This overrides the `merge.renormalize` configuration variable.

no-renames

Turn off rename detection. See also git-diff[1] `--no-renames`.

find-renames[=<n>]

Turn on rename detection, optionally setting the similarity threshold. This is the default. See also git-diff[1] `--find-renames`.

rename-threshold=<n>

Deprecated synonym for `find-renames=<n>`.

subtree[=<path>]

This option is a more advanced form of *subtree* strategy, where the strategy makes a guess on how two trees must be shifted to match with each other when merging. Instead, the specified path is prefixed (or stripped from the beginning) to make the shape of two trees to match.

octopus

This resolves cases with more than two heads, but refuses to do a complex merge that needs manual resolution. It is primarily meant to be used for bundling topic branch heads together. This is the default merge strategy when pulling or merging more than one branch.

ours

This resolves any number of heads, but the resulting tree of the merge is always that of the current branch head, effectively ignoring all changes from all other branches. It is meant to be used to supersede old development history of side branches. Note that this is different from the -Xours option to the *recursive* merge strategy.

subtree

This is a modified recursive strategy. When merging trees A and B, if B corresponds to a subtree of A, B is first adjusted to match the tree structure of A, instead of reading the trees at the same level. This adjustment is also done to the common ancestor tree.

With the strategies that use 3-way merge (including the default, *recursive*), if a change is made on both branches, but later reverted on one of the branches, that change will be present in the merged result; some people find this behavior confusing. It occurs because

only the heads and the merge base are considered when performing a merge, not the individual commits. The merge algorithm therefore considers the reverted change as no change at all, and substitutes the changed version instead.

NOTES

You should understand the implications of using *git rebase* on a repository that you share. See also RECOVERING FROM UPSTREAM REBASE below.

When the git-rebase command is run, it will first execute a "pre-rebase" hook if one exists. You can use this hook to do sanity checks and reject the rebase if it isn’t appropriate. Please see the template pre-rebase hook script for an example.

Upon completion, <branch> will be the current branch.

INTERACTIVE MODE

Rebasing interactively means that you have a chance to edit the commits which are rebased. You can reorder the commits, and you can remove them (weeding out bad or otherwise unwanted patches).

The interactive mode is meant for this type of workflow:

1. have a wonderful idea
2. hack on the code
3. prepare a series for submission
4. submit

where point 2. consists of several instances of

a) regular use

1. finish something worthy of a commit
2. commit

b) independent fixup

1. realize that something does not work
2. fix that
3. commit it

Sometimes the thing fixed in b.2. cannot be amended to the not-quite perfect commit it fixes, because that commit is buried deeply in a patch series. That is exactly what interactive rebase is for: use it after plenty of "a"s and "b"s, by rearranging and editing commits, and squashing multiple commits into one.

Start it with the last commit you want to retain as-is:

```
git rebase -i <after-this-commit>
```

An editor will be fired up with all the commits in your current branch (ignoring merge commits), which come after the given commit. You can reorder the commits in this list to your heart’s content, and you can remove them. The list looks more or less like this:

```
pick deadbee The oneline of this commit
pick fa1afe1 The oneline of the next commit
...
```

The oneline descriptions are purely for your pleasure; *git rebase* will not look at them but at the commit names ("deadbee" and "fa1afe1" in this example), so do not delete or edit the names.

By replacing the command "pick" with the command "edit", you can tell *git rebase* to stop after applying that commit, so that you can edit the files and/or the commit message, amend the commit, and continue rebasing.

If you just want to edit the commit message for a commit, replace the command "pick" with the command "reword".

To drop a commit, replace the command "pick" with "drop", or just delete the matching line.

If you want to fold two or more commits into one, replace the command "pick" for the second and subsequent commits with "squash" or "fixup". If the commits had different authors, the folded commit will be attributed to the author of the first commit. The suggested commit message for the folded commit is the concatenation of the commit messages of the first commit and of those with the "squash" command, but omits the commit messages of commits with the "fixup" command.

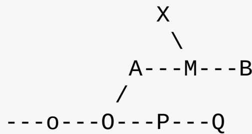
git rebase will stop when "pick" has been replaced with "edit" or when a command fails due to merge errors. When you are done editing and/or resolving conflicts you can continue with `git rebase --continue`.

For example, if you want to reorder the last 5 commits, such that what was HEAD~4 becomes the new HEAD. To achieve that, you would call *git rebase* like this:

```
$ git rebase -i HEAD~5
```

And move the first patch to the end of the list.

You might want to preserve merges, if you have a history like this:



```
           X
            \
         A---M---B
        /
---o---O---P---Q
```

Suppose you want to rebase the side branch starting at "A" to "Q". Make sure that the current HEAD is "B", and call

```
$ git rebase -i -p --onto Q O
```

Reordering and editing commits usually creates untested intermediate steps. You may want to check that your history editing did not break anything by running a test, or at least recompiling at intermediate points in history by using the "exec" command (shortcut "x"). You may do so by creating a todo list like this one:

```
pick deadbee Implement feature XXX
fixup f1a5c00 Fix to feature XXX
exec make
pick c0ffeee The oneline of the next commit
edit deadbab The oneline of the commit after
exec cd subdir; make test
...
```

The interactive rebase will stop when a command fails (i.e. exits with non-0 status) to give you an opportunity to fix the problem. You can continue with `git rebase --continue`.

The "exec" command launches the command in a shell (the one specified in `$SHELL`, or the default shell if `$SHELL` is not set), so you can use shell features (like "cd", ">", ";" …). The command is run from the root of the working tree.

```
$ git rebase -i --exec "make test"
```

This command lets you check that intermediate commits are compilable. The todo list becomes like that:

```
pick 5928aea one
exec make test
pick 04d0fda two
exec make test
pick ba46169 three
exec make test
pick f4593f9 four
exec make test
```

SPLITTING COMMITS

In interactive mode, you can mark commits with the action "edit". However, this does not necessarily mean that *git rebase* expects the result of this edit to be exactly one commit. Indeed, you can undo the commit, or you can add other commits. This can be used to split a commit into two:

- Start an interactive rebase with `git rebase -i <commit>^`, where <commit> is the commit you want to split. In fact, any commit range will do, as long as it contains that commit.
- Mark the commit you want to split with the action "edit".
- When it comes to editing that commit, execute `git reset HEAD^`. The effect is that the HEAD is rewound by one, and the index follows suit. However, the working tree stays the same.
- Now add the changes to the index that you want to have in the first commit. You can use `git add` (possibly interactively) or *git gui* (or both) to do that.
- Commit the now-current index with whatever commit message is appropriate now.
- Repeat the last two steps until your working tree is clean.
- Continue the rebase with `git rebase --continue`.

If you are not absolutely sure that the intermediate revisions are consistent (they compile, pass the testsuite, etc.) you should use *git stash* to stash away the not-yet-committed changes after each commit, test, and amend the commit if fixes are necessary.

RECOVERING FROM UPSTREAM REBASE

Rebasing (or any other form of rewriting) a branch that others have based work on is a bad idea: anyone downstream of it is forced to manually fix their history. This section explains how to do the fix from the downstream’s point of view. The real fix, however, would be to avoid rebasing the upstream in the first place.

To illustrate, suppose you are in a situation where someone develops a *subsystem* branch, and you are working on a *topic* that is dependent on this *subsystem*. You might end up with a history like the following:

```
    o---o---o---o---o---o---o---o---o  master
     \
      o---o---o---o---o  subsystem
                \
                 *---*---*  topic
```

If *subsystem* is rebased against *master*, the following happens:

```
    o---o---o---o---o---o---o---o  master
     \             \
      o---o---o---o---o      o'--o'--o'--o'--o'  subsystem
                \
                 *---*---*  topic
```

If you now continue development as usual, and eventually merge *topic* to *subsystem*, the commits from *subsystem* will remain duplicated forever:

```
    o---o---o---o---o---o---o---o  master
     \             \
      o---o---o---o---o      o'--o'--o'--o'--o'--M     subsystem
                \                         /
                 *---*---*-..........-*--*  topic
```

Such duplicates are generally frowned upon because they clutter up history, making it harder to follow. To clean things up, you need to transplant the commits on *topic* to the new *subsystem* tip, i.e., rebase *topic*. This becomes a ripple effect: anyone downstream from *topic* is forced to rebase too, and so on!

There are two kinds of fixes, discussed in the following subsections:

Easy case: The changes are literally the same.

This happens if the *subsystem* rebase was a simple rebase and had no conflicts.

Hard case: The changes are not the same.

This happens if the *subsystem* rebase had conflicts, or used `--interactive` to omit, edit, squash, or fixup commits; or if the upstream used one of `commit --amend`, `reset`, or `filter-branch`.

The easy case

Only works if the changes (patch IDs based on the diff contents) on *subsystem* are literally the same before and after the rebase *subsystem* did.

In that case, the fix is easy because *git rebase* knows to skip changes that are already present in the new upstream. So if you say (assuming you’re on *topic*)

```
$ git rebase subsystem
```

you will end up with the fixed history

```
o---o---o---o---o---o---o---o  master
             \
              o'--o'--o'--o'--o'  subsystem
                       \
                        *---*---*  topic
```

The hard case

Things get more complicated if the *subsystem* changes do not exactly correspond to the ones before the rebase.

> Note
>
> While an "easy case recovery" sometimes appears to be successful even in the hard case, it may have unintended consequences. For example, a commit that was removed via `git rebase --interactive` will be **resurrected**!

The idea is to manually tell *git rebase* "where the old *subsystem* ended and your *topic* began", that is, what the old merge-base between them was. You will have to find a way to name the last commit of the old *subsystem*, for example:

- With the *subsystem* reflog: after *git fetch*, the old tip of *subsystem* is at `subsystem@{1}`. Subsequent fetches will increase the number. (See git-reflog[1].)
- Relative to the tip of *topic*: knowing that your *topic* has three commits, the old tip of *subsystem* must be `topic~3`.

You can then transplant the old `subsystem..topic` to the new tip by saying (for the reflog case, and assuming you are on *topic* already):

```
$ git rebase --onto subsystem subsystem@{1}
```

The ripple effect of a "hard case" recovery is especially bad: *everyone* downstream from *topic* will now have to perform a "hard case" recovery too!

BUGS

The todo list presented by `--preserve-merges --interactive` does not represent the topology of the revision graph. Editing commits and rewording their commit messages should work fine, but attempts to reorder commits tend to produce counterintuitive results.

For example, an attempt to rearrange

```
1 --- 2 --- 3 --- 4 --- 5
```

to

```
1 --- 2 --- 4 --- 3 --- 5
```

by moving the "pick 4" line will result in the following history:

```
    3
       /
1 --- 2 --- 4 --- 5
```

GIT

Part of the git[1] suite

revert

NAME

git-revert - Revert some existing commits

SYNOPSIS

```
git revert [--[no-]edit] [-n] [-m parent-number] [-s] [-S[<keyid>]] <commit>…
git revert --continue
git revert --quit
git revert --abort
```

DESCRIPTION

Given one or more existing commits, revert the changes that the related patches introduce, and record some new commits that record them. This requires your working tree to be clean (no modifications from the HEAD commit).

Note: *git revert* is used to record some new commits to reverse the effect of some earlier commits (often only a faulty one). If you want to throw away all uncommitted changes in your working directory, you should see git-reset[1], particularly the *--hard* option. If you want to extract specific files as they were in another commit, you should see git-checkout[1], specifically the `git checkout <commit> -- <filename>` syntax. Take care with these alternatives as both will discard uncommitted changes in your working directory.

OPTIONS

<commit>…

Commits to revert. For a more complete list of ways to spell commit names, see gitrevisions[7]. Sets of commits can also be given but no traversal is done by default, see git-rev-list[1] and its *--no-walk* option.

-e

--edit

With this option, *git revert* will let you edit the commit message prior to committing the revert. This is the default if you run the command from a terminal.

-m parent-number

--mainline parent-number

Usually you cannot revert a merge because you do not know which side of the merge should be considered the mainline. This option specifies the parent number (starting from 1) of the mainline and allows revert to reverse the change relative to the specified parent.

Reverting a merge commit declares that you will never want the tree changes brought in by the merge. As a result, later merges will only bring in tree changes introduced by commits that are not ancestors of the previously reverted merge. This may or may not be what you want.

See the revert-a-faulty-merge How-To for more details.

--no-edit

With this option, *git revert* will not start the commit message editor.

-n

--no-commit

Usually the command automatically creates some commits with commit log messages stating which commits were reverted. This flag applies the changes necessary to revert the named commits to your working tree and the index, but does not make the commits. In addition, when this option is used, your index does not have to match the HEAD commit. The revert is done against the beginning state of your index.

This is useful when reverting more than one commits' effect to your index in a row.

-S[<keyid>]

--gpg-sign[=<keyid>]

GPG-sign commits. The `keyid` argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space.

-s

--signoff

Add Signed-off-by line at the end of the commit message. See the signoff option in git-commit[1] for more information.

--strategy=<strategy>

Use the given merge strategy. Should only be used once. See the MERGE STRATEGIES section in git-merge[1] for details.

-X<option>

--strategy-option=<option>

Pass the merge strategy-specific option through to the merge strategy. See git-merge[1] for details.

SEQUENCER SUBCOMMANDS

--continue

Continue the operation in progress using the information in *.git/sequencer*. Can be used to continue after resolving conflicts in a failed cherry-pick or revert.

--quit

Forget about the current operation in progress. Can be used to clear the sequencer state after a failed cherry-pick or revert.

--abort

Cancel the operation and return to the pre-sequence state.

EXAMPLES

`git revert HEAD~3`

Revert the changes specified by the fourth last commit in HEAD and create a new commit with the reverted changes.

`git revert -n master~5..master~2`

Revert the changes done by commits from the fifth last commit in master (included) to the third last commit in master (included), but do not create any commit with the reverted changes. The revert only modifies the working tree and the index.

SEE ALSO

git-cherry-pick[1]

GIT

Part of the git[1] suite

Debugging

bisect

NAME

git-bisect - Use binary search to find the commit that introduced a bug

SYNOPSIS

```
git bisect <subcommand> <options>
```

DESCRIPTION

The command takes various subcommands, and different options depending on the subcommand:

```
git bisect start [--term-{old,good}=<term> --term-{new,bad}=<term>]
         [--no-checkout] [<bad> [<good>...]] [--] [<paths>...]
git bisect (bad|new) [<rev>]
git bisect (good|old) [<rev>...]
git bisect terms [--term-good | --term-bad]
git bisect skip [(<rev>|<range>)...]
git bisect reset [<commit>]
git bisect visualize
git bisect replay <logfile>
git bisect log
git bisect run <cmd>...
git bisect help
```

This command uses a binary search algorithm to find which commit in your project's history introduced a bug. You use it by first telling it a "bad" commit that is known to contain the bug, and a "good" commit that is known to be before the bug was introduced. Then `git bisect` picks a commit between those two endpoints and asks you whether the selected commit is "good" or "bad". It continues narrowing down the range until it finds the exact commit that introduced the change.

In fact, `git bisect` can be used to find the commit that changed **any** property of your project; e.g., the commit that fixed a bug, or the commit that caused a benchmark's performance to improve. To support this more general usage, the terms "old" and "new" can be used in place of "good" and "bad", or you can choose your own terms. See section "Alternate terms" below for more information.

Basic bisect commands: start, bad, good

As an example, suppose you are trying to find the commit that broke a feature that was known to work in version `v2.6.13-rc2` of your project. You start a bisect session as follows:

```
$ git bisect start
$ git bisect bad                 # Current version is bad
$ git bisect good v2.6.13-rc2    # v2.6.13-rc2 is known to be good
```

Once you have specified at least one bad and one good commit, `git bisect` selects a commit in the middle of that range of history, checks it out, and outputs something similar to the following:

```
Bisecting: 675 revisions left to test after this (roughly 10 steps)
```

You should now compile the checked-out version and test it. If that version works correctly, type

```
$ git bisect good
```

If that version is broken, type

```
$ git bisect bad
```

Then `git bisect` will respond with something like

```
Bisecting: 337 revisions left to test after this (roughly 9 steps)
```

Keep repeating the process: compile the tree, test it, and depending on whether it is good or bad run `git bisect good` or `git bisect bad` to ask for the next commit that needs testing.

Eventually there will be no more revisions left to inspect, and the command will print out a description of the first bad commit. The reference `refs/bisect/bad` will be left pointing at that commit.

Bisect reset

After a bisect session, to clean up the bisection state and return to the original HEAD, issue the following command:

```
$ git bisect reset
```

By default, this will return your tree to the commit that was checked out before `git bisect start`. (A new `git bisect start` will also do that, as it cleans up the old bisection state.)

With an optional argument, you can return to a different commit instead:

```
$ git bisect reset <commit>
```

For example, `git bisect reset bisect/bad` will check out the first bad revision, while `git bisect reset HEAD` will leave you on the current bisection commit and avoid switching commits at all.

Alternate terms

Sometimes you are not looking for the commit that introduced a breakage, but rather for a commit that caused a change between some other "old" state and "new" state. For example, you might be looking for the commit that introduced a particular fix. Or you might be looking for the first commit in which the source-code filenames were finally all converted to your company’s naming standard. Or whatever.

In such cases it can be very confusing to use the terms "good" and "bad" to refer to "the state before the change" and "the state after the change". So instead, you can use the terms "old" and "new", respectively, in place of "good" and "bad". (But note that you cannot mix "good" and "bad" with "old" and "new" in a single session.)

In this more general usage, you provide `git bisect` with a "new" commit has some property and an "old" commit that doesn’t have that property. Each time `git bisect` checks out a commit, you test if that commit has the property. If it does, mark the commit as "new"; otherwise, mark it as "old". When the bisection is done, `git bisect` will report which commit introduced the property.

To use "old" and "new" instead of "good" and bad, you must run `git bisect start` without commits as argument and then run the following commands to add the commits:

```
git bisect old [<rev>]
```

to indicate that a commit was before the sought change, or

```
git bisect new [<rev>...]
```

to indicate that it was after.

To get a reminder of the currently used terms, use

```
git bisect terms
```

You can get just the old (respectively new) term with `git bisect term --term-old` or `git bisect term --term-good`.

If you would like to use your own terms instead of "bad"/"good" or "new"/"old", you can choose any names you like (except existing bisect subcommands like `reset`, `start`, ...) by starting the bisection using

```
git bisect start --term-old <term-old> --term-new <term-new>
```

For example, if you are looking for a commit that introduced a performance regression, you might use

```
git bisect start --term-old fast --term-new slow
```

Or if you are looking for the commit that fixed a bug, you might use

```
git bisect start --term-new fixed --term-old broken
```

Then, use `git bisect <term-old>` and `git bisect <term-new>` instead of `git bisect good` and `git bisect bad` to mark commits.

Bisect visualize

To see the currently remaining suspects in *gitk*, issue the following command during the bisection process:

```
$ git bisect visualize
```

`view` may also be used as a synonym for `visualize`.

If the *DISPLAY* environment variable is not set, *git log* is used instead. You can also give command-line options such as `-p` and `--stat`.

```
$ git bisect view --stat
```

Bisect log and bisect replay

After having marked revisions as good or bad, issue the following command to show what has been done so far:

```
$ git bisect log
```

If you discover that you made a mistake in specifying the status of a revision, you can save the output of this command to a file, edit it to remove the incorrect entries, and then issue the following commands to return to a corrected state:

```
$ git bisect reset
$ git bisect replay that-file
```

Avoiding testing a commit

If, in the middle of a bisect session, you know that the suggested revision is not a good one to test (e.g. it fails to build and you know that the failure does not have anything to do with the bug you are chasing), you can manually select a nearby commit and test that one instead.

For example:

```
$ git bisect good/bad            # previous round was good or bad.
Bisecting: 337 revisions left to test after this (roughly 9 steps)
$ git bisect visualize            # oops, that is uninteresting.
$ git reset --hard HEAD~3        # try 3 revisions before what
                    # was suggested
```

Then compile and test the chosen revision, and afterwards mark the revision as good or bad in the usual manner.

Bisect skip

Instead of choosing a nearby commit by yourself, you can ask Git to do it for you by issuing the command:

```
$ git bisect skip                 # Current version cannot be tested
```

However, if you skip a commit adjacent to the one you are looking for, Git will be unable to tell exactly which of those commits was the first bad one.

You can also skip a range of commits, instead of just one commit, using range notation. For example:

```
$ git bisect skip v2.5..v2.6
```

This tells the bisect process that no commit after `v2.5`, up to and including `v2.6`, should be tested.

Note that if you also want to skip the first commit of the range you would issue the command:

```
$ git bisect skip v2.5 v2.5..v2.6
```

This tells the bisect process that the commits between `v2.5` and `v2.6` (inclusive) should be skipped.

Cutting down bisection by giving more parameters to bisect start

You can further cut down the number of trials, if you know what part of the tree is involved in the problem you are tracking down, by specifying path parameters when issuing the `bisect start` command:

```
$ git bisect start -- arch/i386 include/asm-i386
```

If you know beforehand more than one good commit, you can narrow the bisect space down by specifying all of the good commits immediately after the bad commit when issuing the `bisect start` command:

```
$ git bisect start v2.6.20-rc6 v2.6.20-rc4 v2.6.20-rc1 --
                   # v2.6.20-rc6 is bad
                   # v2.6.20-rc4 and v2.6.20-rc1 are good
```

Bisect run

If you have a script that can tell if the current source code is good or bad, you can bisect by issuing the command:

```
$ git bisect run my_script arguments
```

Note that the script (`my_script` in the above example) should exit with code 0 if the current source code is good/old, and exit with a code between 1 and 127 (inclusive), except 125, if the current source code is bad/new.

Any other exit code will abort the bisect process. It should be noted that a program that terminates via `exit(-1)` leaves $? = 255, (see the exit(3) manual page), as the value is chopped with `& 0377`.

The special exit code 125 should be used when the current source code cannot be tested. If the script exits with this code, the current revision will be skipped (see `git bisect skip` above). 125 was chosen as the highest sensible value to use for this purpose, because 126 and 127 are used by POSIX shells to signal specific error status (127 is for command not found, 126 is for command found but not executable—these details do not matter, as they are normal errors in the script, as far as `bisect run` is concerned).

You may often find that during a bisect session you want to have temporary modifications (e.g. s/#define DEBUG 0/#define DEBUG 1/ in a header file, or "revision that does not have this commit needs this patch applied to work around another problem this bisection is not interested in") applied to the revision being tested.

To cope with such a situation, after the inner *git bisect* finds the next revision to test, the script can apply the patch before compiling, run the real test, and afterwards decide if the revision (possibly with the needed patch) passed the test and then rewind the tree to the pristine state. Finally the script should exit with the status of the real test to let the `git bisect run` command loop determine the eventual outcome of the bisect session.

OPTIONS

--no-checkout

Do not checkout the new working tree at each iteration of the bisection process. Instead just update a special reference named *BISECT_HEAD* to make it point to the commit that should be tested.

This option may be useful when the test you would perform in each step does not require a checked out tree.

If the repository is bare, `--no-checkout` is assumed.

EXAMPLES

- Automatically bisect a broken build between v1.2 and HEAD:

```
$ git bisect start HEAD v1.2 --      # HEAD is bad, v1.2 is good
$ git bisect run make                # "make" builds the app
$ git bisect reset                   # quit the bisect session
```

- Automatically bisect a test failure between origin and HEAD:

```
$ git bisect start HEAD origin --    # HEAD is bad, origin is good
$ git bisect run make test           # "make test" builds and tests
$ git bisect reset                   # quit the bisect session
```

- Automatically bisect a broken test case:

```
$ cat ~/test.sh
#!/bin/sh
make || exit 125                     # this skips broken builds
~/check_test_case.sh                 # does the test case pass?
$ git bisect start HEAD HEAD~10 --   # culprit is among the last 10
$ git bisect run ~/test.sh
$ git bisect reset                   # quit the bisect session
```

Here we use a `test.sh` custom script. In this script, if `make` fails, we skip the current commit. `check_test_case.sh` should `exit 0` if the test case passes, and `exit 1` otherwise.

It is safer if both `test.sh` and `check_test_case.sh` are outside the repository to prevent interactions between the bisect, make and test processes and the scripts.

- Automatically bisect with temporary modifications (hot-fix):

```
$ cat ~/test.sh
#!/bin/sh

# tweak the working tree by merging the hot-fix branch
# and then attempt a build
if    git merge --no-commit hot-fix &&
    make
then
    # run project specific test and report its status
    ~/check_test_case.sh
    status=$?
else
    # tell the caller this is untestable
    status=125
fi

# undo the tweak to allow clean flipping to the next commit
git reset --hard

# return control
exit $status
```

This applies modifications from a hot-fix branch before each test run, e.g. in case your build or test environment changed so that older revisions may need a fix which newer ones have already. (Make sure the hot-fix branch is based off a commit which is contained in all revisions which you are bisecting, so that the merge does not pull in too much, or use `git cherry-pick` instead of `git merge`.)

- Automatically bisect a broken test case:

```
$ git bisect start HEAD HEAD~10 --   # culprit is among the last 10
$ git bisect run sh -c "make || exit 125; ~/check_test_case.sh"
$ git bisect reset                   # quit the bisect session
```

This shows that you can do without a run script if you write the test on a single line.

- Locate a good region of the object graph in a damaged repository

```
$ git bisect start HEAD &lt;known-good-commit&gt; [ &lt;boundary-commit&gt; ... ] --r
$ git bisect run sh -c '
    GOOD=$(git for-each-ref "--format=%(objectname)" refs/bisect/good-*) &&
    git rev-list --objects BISECT_HEAD --not $GOOD &gt;tmp.$$ &&
    git pack-objects --stdout &gt;/dev/null &lt;tmp.$$
    rc=$?
    rm -f tmp.$$
    test $rc = 0'

$ git bisect reset                   # quit the bisect session
```

In this case, when *git bisect run* finishes, bisect/bad will refer to a commit that has at least one parent whose reachable graph is fully traversable in the sense required by *git pack objects*.

- Look for a fix instead of a regression in the code

```
$ git bisect start
$ git bisect new HEAD    # current commit is marked as new
$ git bisect old HEAD~10 # the tenth commit from now is marked as old
```

or:

```
$ git bisect start --term-old broken --term-new fixed
$ git bisect fixed
$ git bisect broken HEAD~10
```

Getting help

Use `git bisect` to get a short usage description, and `git bisect help` or `git bisect -h` to get a long usage description.

SEE ALSO

Fighting regressions with git bisect, git-blame[1].

GIT

Part of the git[1] suite

blame

NAME

git-blame - Show what revision and author last modified each line of a file

SYNOPSIS

```
git blame [-c] [-b] [-l] [--root] [-t] [-f] [-n] [-s] [-e] [-p] [-w] [--incremental]
        [-L <range>] [-S <revs-file>] [-M] [-C] [-C] [-C] [--since=<date>]
        [--progress] [--abbrev=<n>] [<rev> | --contents <file> | --reverse <rev>]
        [--] <file>
```

DESCRIPTION

Annotates each line in the given file with information from the revision which last modified the line. Optionally, start annotating from the given revision.

When specified one or more times, `-L` restricts annotation to the requested lines.

The origin of lines is automatically followed across whole-file renames (currently there is no option to turn the rename-following off). To follow lines moved from one file to another, or to follow lines that were copied and pasted from another file, etc., see the `-C` and `-M` options.

The report does not tell you anything about lines which have been deleted or replaced; you need to use a tool such as *git diff* or the "pickaxe" interface briefly mentioned in the following paragraph.

Apart from supporting file annotation, Git also supports searching the development history for when a code snippet occurred in a change. This makes it possible to track when a code snippet was added to a file, moved or copied between files, and eventually deleted or replaced. It works by searching for a text string in the diff. A small example of the pickaxe interface that searches for `blame_usage` :

```
$ git log --pretty=oneline -S'blame_usage'
5040f17eba15504bad66b14a645bddd9b015ebb7 blame -S <ancestry-file>
ea4c7f9bf69e781dd0cd88d2bccb2bf5cc15c9a7 git-blame: Make the output
```

OPTIONS

-b

Show blank SHA-1 for boundary commits. This can also be controlled via the `blame.blankboundary` config option.

--root

Do not treat root commits as boundaries. This can also be controlled via the `blame.showRoot` config option.

--show-stats

Include additional statistics at the end of blame output.

-L <start>,<end>

-L :<funcname>

Annotate only the given line range. May be specified multiple times. Overlapping ranges are allowed.

<start> and <end> are optional. “-L <start>” or “-L <start>,” spans from <start> to end of file. “-L ,<end>” spans from start of file to <end>.

line-range-format.txt

-l

Show long rev (Default: off).

-t

Show raw timestamp (Default: off).

-S <revs-file>

Use revisions from revs-file instead of calling git-rev-list[1].

--reverse

Walk history forward instead of backward. Instead of showing the revision in which a line appeared, this shows the last revision in which a line has existed. This requires a range of revision like START..END where the path to blame exists in START.

-p

--porcelain

Show in a format designed for machine consumption.

--line-porcelain

Show the porcelain format, but output commit information for each line, not just the first time a commit is referenced. Implies --porcelain.

--incremental

Show the result incrementally in a format designed for machine consumption.

--encoding=<encoding>

Specifies the encoding used to output author names and commit summaries. Setting it to `none` makes blame output unconverted data. For more information see the discussion about encoding in the git-log[1] manual page.

--contents <file>

When <rev> is not specified, the command annotates the changes starting backwards from the working tree copy. This flag makes the command pretend as if the working tree copy has the contents of the named file (specify `-` to make the command read from the standard input).

--date <format>

Specifies the format used to output dates. If --date is not provided, the value of the blame.date config variable is used. If the blame.date config variable is also not set, the iso format is used. For supported values, see the discussion of the --date option at git-log[1].

--[no-]progress

Progress status is reported on the standard error stream by default when it is attached to a terminal. This flag enables progress reporting even if not attached to a terminal. Can't use `--progress` together with `--porcelain` or `--incremental`.

-M|<num>|

Detect moved or copied lines within a file. When a commit moves or copies a block of lines (e.g. the original file has A and then B, and the commit changes it to B and then A), the traditional *blame* algorithm notices only half of the movement and typically blames the lines that were moved up (i.e. B) to the parent and assigns blame to the lines that were moved down (i.e. A) to the child commit. With this option, both groups of lines are blamed on the parent by running extra passes of inspection.

<num> is optional but it is the lower bound on the number of alphanumeric characters that Git must detect as moving/copying within a file for it to associate those lines with the parent commit. The default value is 20.

-C|<num>|

In addition to `-M`, detect lines moved or copied from other files that were modified in the same commit. This is useful when you reorganize your program and move code around across files. When this option is given twice, the command additionally looks for copies from other files in the commit that creates the file. When this option is given three times, the command additionally looks for copies from other files in any commit.

<num> is optional but it is the lower bound on the number of alphanumeric characters that Git must detect as moving/copying between files for it to associate those lines with the parent commit. And the default value is 40. If there are more than one `-C` options given, the <num> argument of the last `-C` will take effect.

-h

Show help message.

-c

Use the same output mode as git-annotate[1] (Default: off).

--score-debug

Include debugging information related to the movement of lines between files (see `-C`) and lines moved within a file (see `-M`). The first number listed is the score. This is the number of alphanumeric characters detected as having been moved between or within files. This must be above a certain threshold for *git blame* to consider those lines of code to have been moved.

-f

--show-name

Show the filename in the original commit. By default the filename is shown if there is any line that came from a file with a different name, due to rename detection.

-n

--show-number

Show the line number in the original commit (Default: off).

-s

Suppress the author name and timestamp from the output.

-e

--show-email

Show the author email instead of author name (Default: off). This can also be controlled via the `blame.showEmail` config option.

-w

Ignore whitespace when comparing the parent’s version and the child’s to find where the lines came from.

--abbrev=<n>

Instead of using the default 7+1 hexadecimal digits as the abbreviated object name, use <n>+1 digits. Note that 1 column is used for a caret to mark the boundary commit.

THE PORCELAIN FORMAT

In this format, each line is output after a header; the header at the minimum has the first line which has:

- 40-byte SHA-1 of the commit the line is attributed to;
- the line number of the line in the original file;
- the line number of the line in the final file;
- on a line that starts a group of lines from a different commit than the previous one, the number of lines in this group. On subsequent lines this field is absent.

This header line is followed by the following information at least once for each commit:

- the author name ("author"), email ("author-mail"), time ("author-time"), and time zone ("author-tz"); similarly for committer.
- the filename in the commit that the line is attributed to.
- the first line of the commit log message ("summary").

The contents of the actual line is output after the above header, prefixed by a TAB. This is to allow adding more header elements later.

The porcelain format generally suppresses commit information that has already been seen. For example, two lines that are blamed to the same commit will both be shown, but the details for that commit will be shown only once. This is more efficient, but may require more state be kept by the reader. The `--line-porcelain` option can be used to output full commit information for each line, allowing simpler (but less efficient) usage like:

```
# count the number of lines attributed to each author
git blame --line-porcelain file |
sed -n 's/^author //p' |
sort | uniq -c | sort -rn
```

SPECIFYING RANGES

Unlike *git blame* and *git annotate* in older versions of git, the extent of the annotation can be limited to both line ranges and revision ranges. The `-L` option, which limits annotation to a range of lines, may be specified multiple times.

When you are interested in finding the origin for lines 40-60 for file `foo`, you can use the `-L` option like so (they mean the same thing — both ask for 21 lines starting at line 40):

```
git blame -L 40,60 foo
git blame -L 40,+21 foo
```

Also you can use a regular expression to specify the line range:

```
git blame -L '/^sub hello {/,/^}$/' foo
```

which limits the annotation to the body of the `hello` subroutine.

When you are not interested in changes older than version v2.6.18, or changes older than 3 weeks, you can use revision range specifiers similar to *git rev-list*:

```
git blame v2.6.18.. -- foo
git blame --since=3.weeks -- foo
```

When revision range specifiers are used to limit the annotation, lines that have not changed since the range boundary (either the commit v2.6.18 or the most recent commit that is more than 3 weeks old in the above example) are blamed for that range boundary commit.

A particularly useful way is to see if an added file has lines created by copy-and-paste from existing files. Sometimes this indicates that the developer was being sloppy and did not refactor the code properly. You can first find the commit that introduced the file with:

```
git log --diff-filter=A --pretty=short -- foo
```

and then annotate the change between the commit and its parents, using `commit^!` notation:

```
git blame -C -C -f $commit^! -- foo
```

INCREMENTAL OUTPUT

When called with `--incremental` option, the command outputs the result as it is built. The output generally will talk about lines touched by more recent commits first (i.e. the lines will be annotated out of order) and is meant to be used by interactive viewers.

The output format is similar to the Porcelain format, but it does not contain the actual lines from the file that is being annotated.

1. Each blame entry always starts with a line of:

   ```
   &lt;40-byte hex sha1&gt; &lt;sourceline&gt; &lt;resultline&gt; &lt;num_lines&gt;
   ```

   Line numbers count from 1.

2. The first time that a commit shows up in the stream, it has various other information about it printed out with a one-word tag at the beginning of each line describing the extra commit information (author, email, committer, dates, summary, etc.).

3. Unlike the Porcelain format, the filename information is always given and terminates the entry:

   ```
   "filename" &lt;whitespace-quoted-filename-goes-here&gt;
   ```

   and thus it is really quite easy to parse for some line- and word-oriented parser (which should be quite natural for most scripting languages).

   | Note | For people who do parsing: to make it more robust, just ignore any lines between the first and last one ("<sha1>" and "filename" lines) where you do not recognize the tag words (or care about that particular one) at the beginning of the "extended information" lines. That way, if there is ever added information (like the commit encoding or extended commit commentary), a blame viewer will not care. |

MAPPING AUTHORS

If the file `.mailmap` exists at the toplevel of the repository, or at the location pointed to by the mailmap.file or mailmap.blob configuration options, it is used to map author and committer names and email addresses to canonical real names and email addresses.

In the simple form, each line in the file consists of the canonical real name of an author, whitespace, and an email address used in the commit (enclosed by < and >) to map to the name. For example:

```
Proper Name <commit@email.xx>
```

The more complex forms are:

```
<proper@email.xx> <commit@email.xx>
```

which allows mailmap to replace only the email part of a commit, and:

```
Proper Name <proper@email.xx> <commit@email.xx>
```

which allows mailmap to replace both the name and the email of a commit matching the specified commit email address, and:

```
Proper Name <proper@email.xx> Commit Name <commit@email.xx>
```

which allows mailmap to replace both the name and the email of a commit matching both the specified commit name and email address.

Example 1: Your history contains commits by two authors, Jane and Joe, whose names appear in the repository under several forms:

```
Joe Developer <joe@example.com>
Joe R. Developer <joe@example.com>
Jane Doe <jane@example.com>
Jane Doe <jane@laptop.(none)>
Jane D. <jane@desktop.(none)>
```

Now suppose that Joe wants his middle name initial used, and Jane prefers her family name fully spelled out. A proper `.mailmap` file would look like:

```
Jane Doe         <jane@desktop.(none)>
Joe R. Developer <joe@example.com>
```

Note how there is no need for an entry for `<jane@laptop.(none)>` , because the real name of that author is already correct.

Example 2: Your repository contains commits from the following authors:

```
nick1 <bugs@company.xx>
nick2 <bugs@company.xx>
nick2 <nick2@company.xx>
santa <me@company.xx>
claus <me@company.xx>
CTO <cto@coompany.xx>
```

Then you might want a `.mailmap` file that looks like:

```
<cto@company.xx>                       <cto@coompany.xx>
Some Dude <some@dude.xx>         nick1 <bugs@company.xx>
Other Author <other@author.xx>   nick2 <bugs@company.xx>
Other Author <other@author.xx>         <nick2@company.xx>
Santa Claus <santa.claus@northpole.xx> <me@company.xx>
```

Use hash # for comments that are either on their own line, or after the email address.

SEE ALSO

git-annotate[1]

GIT

Part of the git[1] suite

grep

NAME

git-grep - Print lines matching a pattern

SYNOPSIS

```
git grep [-a | --text] [-I] [--textconv] [-i | --ignore-case] [-w | --word-regexp]
       [-v | --invert-match] [-h|-H] [--full-name]
       [-E | --extended-regexp] [-G | --basic-regexp]
       [-P | --perl-regexp]
       [-F | --fixed-strings] [-n | --line-number]
       [-l | --files-with-matches] [-L | --files-without-match]
       [(-O | --open-files-in-pager) [<pager>]]
       [-z | --null]
       [-c | --count] [--all-match] [-q | --quiet]
       [--max-depth <depth>]
       [--color[=<when>] | --no-color]
       [--break] [--heading] [-p | --show-function]
       [-A <post-context>] [-B <pre-context>] [-C <context>]
       [-W | --function-context]
       [--threads <num>]
       [-f <file>] [-e] <pattern>
       [--and|--or|--not|(|)|-e <pattern>…]
       [ [--[no-]exclude-standard] [--cached | --no-index | --untracked] | <tree>…]
       [--] [<pathspec>…]
```

DESCRIPTION

Look for specified patterns in the tracked files in the work tree, blobs registered in the index file, or blobs in given tree objects. Patterns are lists of one or more search expressions separated by newline characters. An empty string as search expression matches all lines.

CONFIGURATION

grep.lineNumber

If set to true, enable *-n* option by default.

grep.patternType

Set the default matching behavior. Using a value of *basic*, *extended*, *fixed*, or *perl* will enable the *--basic-regexp*, *--extended-regexp*, *--fixed-strings*, or *--perl-regexp* option accordingly, while the value *default* will return to the default matching behavior.

grep.extendedRegexp

If set to true, enable *--extended-regexp* option by default. This option is ignored when the *grep.patternType* option is set to a value other than *default*.

grep.threads

Number of grep worker threads to use. If unset (or set to 0), 8 threads are used by default (for now).

grep.fullName

If set to true, enable *--full-name* option by default.

grep.fallbackToNoIndex

If set to true, fall back to git grep --no-index if git grep is executed outside of a git repository. Defaults to false.

OPTIONS

--cached

Instead of searching tracked files in the working tree, search blobs registered in the index file.

--no-index

Search files in the current directory that is not managed by Git.

--untracked

In addition to searching in the tracked files in the working tree, search also in untracked files.

--no-exclude-standard

Also search in ignored files by not honoring the `.gitignore` mechanism. Only useful with `--untracked`.

--exclude-standard

Do not pay attention to ignored files specified via the `.gitignore` mechanism. Only useful when searching files in the current directory with `--no-index`.

-a

--text

Process binary files as if they were text.

--textconv

Honor textconv filter settings.

--no-textconv

Do not honor textconv filter settings. This is the default.

-i

--ignore-case

Ignore case differences between the patterns and the files.

-I

Don’t match the pattern in binary files.

--max-depth <depth>

For each <pathspec> given on command line, descend at most <depth> levels of directories. A negative value means no limit. This option is ignored if <pathspec> contains active wildcards. In other words if "a*" matches a directory named "a*", "*" is matched literally so --max-depth is still effective.

-w

--word-regexp

Match the pattern only at word boundary (either begin at the beginning of a line, or preceded by a non-word character; end at the end of a line or followed by a non-word character).

-v

--invert-match

Select non-matching lines.

-h

-H

By default, the command shows the filename for each match. `-h` option is used to suppress this output. `-H` is there for completeness and does not do anything except it overrides `-h` given earlier on the command line.

--full-name

When run from a subdirectory, the command usually outputs paths relative to the current directory. This option forces paths to be output relative to the project top directory.

-E

--extended-regexp

-G

--basic-regexp

Use POSIX extended/basic regexp for patterns. Default is to use basic regexp.

-P

--perl-regexp

Use Perl-compatible regexp for patterns. Requires libpcre to be compiled in.

-F

--fixed-strings

Use fixed strings for patterns (don’t interpret pattern as a regex).

-n

--line-number

Prefix the line number to matching lines.

-l

--files-with-matches

--name-only

-L

--files-without-match

Instead of showing every matched line, show only the names of files that contain (or do not contain) matches. For better compatibility with *git diff*, `--name-only` is a synonym for `--files-with-matches`.

-O[<pager>]

--open-files-in-pager[=<pager>]

Open the matching files in the pager (not the output of *grep*). If the pager happens to be "less" or "vi", and the user specified only one pattern, the first file is positioned at the first match automatically. The `pager` argument is optional; if specified, it must be stuck to the option without a space. If `pager` is unspecified, the default pager will be used (see `core.pager` in git-config[1]).

-z

--null

Output \0 instead of the character that normally follows a file name.

-c

--count

Instead of showing every matched line, show the number of lines that match.

--color[=<when>]

Show colored matches. The value must be always (the default), never, or auto.

--no-color

Turn off match highlighting, even when the configuration file gives the default to color output. Same as `--color=never`.

--break

Print an empty line between matches from different files.

--heading

Show the filename above the matches in that file instead of at the start of each shown line.

-p

--show-function

Show the preceding line that contains the function name of the match, unless the matching line is a function name itself. The name is determined in the same way as *git diff* works out patch hunk headers (see *Defining a custom hunk-header* in gitattributes[5]).

-<num>

-C <num>

--context <num>

Show <num> leading and trailing lines, and place a line containing `--` between contiguous groups of matches.

-A <num>

--after-context <num>

Show <num> trailing lines, and place a line containing `--` between contiguous groups of matches.

-B <num>

--before-context <num>

Show <num> leading lines, and place a line containing `--` between contiguous groups of matches.

-W

--function-context

Show the surrounding text from the previous line containing a function name up to the one before the next function name, effectively showing the whole function in which the match was found.

--threads <num>

Number of grep worker threads to use. See `grep.threads` in *CONFIGURATION* for more information.

-f <file>

Read patterns from <file>, one per line.

-e

The next parameter is the pattern. This option has to be used for patterns starting with `-` and should be used in scripts passing user input to grep. Multiple patterns are combined by *or*.

--and

--or

--not

(…)

Specify how multiple patterns are combined using Boolean expressions. `--or` is the default operator. `--and` has higher precedence than `--or`. `-e` has to be used for all patterns.

--all-match

When giving multiple pattern expressions combined with `--or`, this flag is specified to limit the match to files that have lines to match all of them.

-q

--quiet

Do not output matched lines; instead, exit with status 0 when there is a match and with non-zero status when there isn't.

<tree>…

Instead of searching tracked files in the working tree, search blobs in the given trees.

--

Signals the end of options; the rest of the parameters are <pathspec> limiters.

<pathspec>…

If given, limit the search to paths matching at least one pattern. Both leading paths match and glob(7) patterns are supported.

Examples

```
git grep 'time_t' -- '*.[ch]'
```

Looks for `time_t` in all tracked .c and .h files in the working directory and its subdirectories.

```
git grep -e '#define' --and \( -e MAX_PATH -e PATH_MAX \)
```

Looks for a line that has `#define` and either `MAX_PATH` or `PATH_MAX`.

```
git grep --all-match -e NODE -e Unexpected
```

Looks for a line that has `NODE` or `Unexpected` in files that have lines that match both.

GIT

Part of the git[1] suite

Email

am

NAME

git-am - Apply a series of patches from a mailbox

SYNOPSIS

```
git am [--signoff] [--keep] [--[no-]keep-cr] [--[no-]utf8]
     [--[no-]3way] [--interactive] [--committer-date-is-author-date]
     [--ignore-date] [--ignore-space-change | --ignore-whitespace]
     [--whitespace=<option>] [-C<n>] [-p<n>] [--directory=<dir>]
     [--exclude=<path>] [--include=<path>] [--reject] [-q | --quiet]
     [--[no-]scissors] [-S[<keyid>]] [--patch-format=<format>]
     [(<mbox> | <Maildir>)…]
git am (--continue | --skip | --abort)
```

DESCRIPTION

Splits mail messages in a mailbox into commit log message, authorship information and patches, and applies them to the current branch.

OPTIONS

(<mbox>|<Maildir>)…

The list of mailbox files to read patches from. If you do not supply this argument, the command reads from the standard input. If you supply directories, they will be treated as Maildirs.

-s

--signoff

Add a `Signed-off-by:` line to the commit message, using the committer identity of yourself. See the signoff option in git-commit[1] for more information.

-k

--keep

Pass `-k` flag to *git mailinfo* (see git-mailinfo[1]).

--keep-non-patch

Pass `-b` flag to *git mailinfo* (see git-mailinfo[1]).

--[no-]keep-cr

With `--keep-cr`, call *git mailsplit* (see git-mailsplit[1]) with the same option, to prevent it from stripping CR at the end of lines. `am.keepcr` configuration variable can be used to specify the default behaviour. `--no-keep-cr` is useful to override `am.keepcr`.

-c

--scissors

Remove everything in body before a scissors line (see git-mailinfo[1]). Can be activated by default using the `mailinfo.scissors` configuration variable.

--no-scissors

Ignore scissors lines (see git-mailinfo[1]).

-m

--message-id

Pass the `-m` flag to *git mailinfo* (see git-mailinfo[1]), so that the Message-ID header is added to the commit message. The `am.messageid` configuration variable can be used to specify the default behaviour.

--no-message-id

Do not add the Message-ID header to the commit message. `no-message-id` is useful to override `am.messageid`.

-q

--quiet

Be quiet. Only print error messages.

-u

--utf8

Pass `-u` flag to *git mailinfo* (see git-mailinfo[1]). The proposed commit log message taken from the e-mail is re-coded into UTF-8 encoding (configuration variable `i18n.commitencoding` can be used to specify project's preferred encoding if it is not UTF-8).

This was optional in prior versions of git, but now it is the default. You can use `--no-utf8` to override this.

--no-utf8

Pass `-n` flag to *git mailinfo* (see git-mailinfo[1]).

-3

--3way

--no-3way

When the patch does not apply cleanly, fall back on 3-way merge if the patch records the identity of blobs it is supposed to apply to and we have those blobs available locally. `--no-3way` can be used to override am.threeWay configuration variable. For more information, see am.threeWay in git-config[1].

--ignore-space-change

--ignore-whitespace

--whitespace=<option>

-C<n>

-p<n>

--directory=<dir>

--exclude=<path>

--include=<path>

--reject

These flags are passed to the *git apply* (see git-apply[1]) program that applies the patch.

--patch-format

By default the command will try to detect the patch format automatically. This option allows the user to bypass the automatic detection and specify the patch format that the patch(es) should be interpreted as. Valid formats are mbox, stgit, stgit-series and hg.

-i

--interactive

Run interactively.

--committer-date-is-author-date

By default the command records the date from the e-mail message as the commit author date, and uses the time of commit creation as the committer date. This allows the user to lie about the committer date by using the same value as the author date.

--ignore-date

By default the command records the date from the e-mail message as the commit author date, and uses the time of commit creation as the committer date. This allows the user to lie about the author date by using the same value as the committer date.

--skip

Skip the current patch. This is only meaningful when restarting an aborted patch.

-S[<keyid>]

--gpg-sign[=<keyid>]

GPG-sign commits. The `keyid` argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space.

--continue

-r

--resolved

After a patch failure (e.g. attempting to apply conflicting patch), the user has applied it by hand and the index file stores the result of the application. Make a commit using the authorship and commit log extracted from the e-mail message and the current index file, and continue.

--resolvemsg=<msg>

When a patch failure occurs, <msg> will be printed to the screen before exiting. This overrides the standard message informing you to use `--continue` or `--skip` to handle the failure. This is solely for internal use between *git rebase* and *git am*.

--abort

Restore the original branch and abort the patching operation.

DISCUSSION

The commit author name is taken from the "From: " line of the message, and commit author date is taken from the "Date: " line of the message. The "Subject: " line is used as the title of the commit, after stripping common prefix "[PATCH <anything>]". The "Subject: " line is

supposed to concisely describe what the commit is about in one line of text.

"From: " and "Subject: " lines starting the body override the respective commit author name and title values taken from the headers.

The commit message is formed by the title taken from the "Subject: ", a blank line and the body of the message up to where the patch begins. Excess whitespace at the end of each line is automatically stripped.

The patch is expected to be inline, directly following the message. Any line that is of the form:

- three-dashes and end-of-line, or
- a line that begins with "diff -", or
- a line that begins with "Index: "

is taken as the beginning of a patch, and the commit log message is terminated before the first occurrence of such a line.

When initially invoking `git am`, you give it the names of the mailboxes to process. Upon seeing the first patch that does not apply, it aborts in the middle. You can recover from this in one of two ways:

1. skip the current patch by re-running the command with the *--skip* option.
2. hand resolve the conflict in the working directory, and update the index file to bring it into a state that the patch should have produced. Then run the command with the *--continue* option.

The command refuses to process new mailboxes until the current operation is finished, so if you decide to start over from scratch, run `git am --abort` before running the command with mailbox names.

Before any patches are applied, ORIG*HEAD is set to the tip of the current branch. This is useful if you have problems with multiple commits, like running _git am* on the wrong branch or an error in the commits that is more easily fixed by changing the mailbox (e.g. errors in the "From:" lines).

HOOKS

This command can run `applypatch-msg`, `pre-applypatch`, and `post-applypatch` hooks. See githooks[5] for more information.

SEE ALSO

git-apply[1].

GIT

Part of the git[1] suite

apply

NAME

git-apply - Apply a patch to files and/or to the index

SYNOPSIS

```
git apply [--stat] [--numstat] [--summary] [--check] [--index] [--3way]
          [--apply] [--no-add] [--build-fake-ancestor=<file>] [-R | --reverse]
          [--allow-binary-replacement | --binary] [--reject] [-z]
          [-p<n>] [-C<n>] [--inaccurate-eof] [--recount] [--cached]
          [--ignore-space-change | --ignore-whitespace ]
          [--whitespace=(nowarn|warn|fix|error|error-all)]
          [--exclude=<path>] [--include=<path>] [--directory=<root>]
          [--verbose] [--unsafe-paths] [<patch>…]
```

DESCRIPTION

Reads the supplied diff output (i.e. "a patch") and applies it to files. With the `--index` option the patch is also applied to the index, and with the `--cached` option the patch is only applied to the index. Without these options, the command applies the patch only to files, and does not require them to be in a Git repository.

This command applies the patch but does not create a commit. Use git-am[1] to create commits from patches generated by git-format-patch[1] and/or received by email.

OPTIONS

<patch>…

The files to read the patch from. - can be used to read from the standard input.

--stat

Instead of applying the patch, output diffstat for the input. Turns off "apply".

--numstat

Similar to `--stat`, but shows the number of added and deleted lines in decimal notation and the pathname without abbreviation, to make it more machine friendly. For binary files, outputs two `-` instead of saying `0 0`. Turns off "apply".

--summary

Instead of applying the patch, output a condensed summary of information obtained from git diff extended headers, such as creations, renames and mode changes. Turns off "apply".

--check

Instead of applying the patch, see if the patch is applicable to the current working tree and/or the index file and detects errors. Turns off "apply".

--index

When `--check` is in effect, or when applying the patch (which is the default when none of the options that disables it is in effect), make sure the patch is applicable to what the current index file records. If the file to be patched in the working tree is not up-to-date, it is flagged as an error. This flag also causes the index file to be updated.

--cached

Apply a patch without touching the working tree. Instead take the cached data, apply the patch, and store the result in the index without using the working tree. This implies `--index`.

-3

--3way

When the patch does not apply cleanly, fall back on 3-way merge if the patch records the identity of blobs it is supposed to apply to, and we have those blobs available locally, possibly leaving the conflict markers in the files in the working tree for the user to resolve. This option implies the `--index` option, and is incompatible with the `--reject` and the `--cached` options.

--build-fake-ancestor=<file>

Newer *git diff* output has embedded *index information* for each blob to help identify the original version that the patch applies to. When this flag is given, and if the original versions of the blobs are available locally, builds a temporary index containing those blobs.

When a pure mode change is encountered (which has no index information), the information is read from the current index instead.

-R

--reverse

Apply the patch in reverse.

--reject

For atomicity, *git apply* by default fails the whole patch and does not touch the working tree when some of the hunks do not apply. This option makes it apply the parts of the patch that are applicable, and leave the rejected hunks in corresponding *.rej files.

-z

When `--numstat` has been given, do not munge pathnames, but use a NUL-terminated machine-readable format.

Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

-p<n>

Remove <n> leading slashes from traditional diff paths. The default is 1.

-C<n>

Ensure at least <n> lines of surrounding context match before and after each change. When fewer lines of surrounding context exist they all must match. By default no context is ever ignored.

--unidiff-zero

By default, *git apply* expects that the patch being applied is a unified diff with at least one line of context. This provides good safety measures, but breaks down when applying a diff generated with `--unified=0`. To bypass these checks use `--unidiff-zero`.

Note, for the reasons stated above usage of context-free patches is discouraged.

--apply

If you use any of the options marked "Turns off *apply*" above, *git apply* reads and outputs the requested information without actually applying the patch. Give this flag after those flags to also apply the patch.

--no-add

When applying a patch, ignore additions made by the patch. This can be used to extract the common part between two files by first running *diff* on them and applying the result with this option, which would apply the deletion part but not the addition part.

--allow-binary-replacement

--binary

Historically we did not allow binary patch applied without an explicit permission from the user, and this flag was the way to do so. Currently we always allow binary patch application, so this is a no-op.

--exclude=<path-pattern>

Don’t apply changes to files matching the given path pattern. This can be useful when importing patchsets, where you want to exclude certain files or directories.

--include=<path-pattern>

Apply changes to files matching the given path pattern. This can be useful when importing patchsets, where you want to include certain files or directories.

When `--exclude` and `--include` patterns are used, they are examined in the order they appear on the command line, and the first match determines if a patch to each path is used. A patch to a path that does not match any include/exclude pattern is used by default if there is no include pattern on the command line, and ignored if there is any include pattern.

--ignore-space-change

--ignore-whitespace

When applying a patch, ignore changes in whitespace in context lines if necessary. Context lines will preserve their whitespace, and they will not undergo whitespace fixing regardless of the value of the `--whitespace` option. New lines will still be fixed, though.

--whitespace=<action>

When applying a patch, detect a new or modified line that has whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors.

By default, the command outputs warning messages but applies the patch. When `git-apply` is used for statistics and not applying a patch, it defaults to `nowarn` .

You can use different `<action>` values to control this behavior:

- `nowarn` turns off the trailing whitespace warning.
- `warn` outputs warnings for a few such errors, but applies the patch as-is (default).
- `fix` outputs warnings for a few such errors, and applies the patch after fixing them (`strip` is a synonym --- the tool used to consider only trailing whitespace characters as errors, and the fix involved *stripping* them, but modern Gits do more).

- `error` outputs warnings for a few such errors, and refuses to apply the patch.
- `error-all` is similar to `error` but shows all errors.

--inaccurate-eof

Under certain circumstances, some versions of *diff* do not correctly detect a missing new-line at the end of the file. As a result, patches created by such *diff* programs do not record incomplete lines correctly. This option adds support for applying such patches by working around this bug.

-v

--verbose

Report progress to stderr. By default, only a message about the current patch being applied will be printed. This option will cause additional information to be reported.

--recount

Do not trust the line counts in the hunk headers, but infer them by inspecting the patch (e.g. after editing the patch without adjusting the hunk headers appropriately).

--directory=<root>

Prepend <root> to all filenames. If a "-p" argument was also passed, it is applied before prepending the new root.

For example, a patch that talks about updating `a/git-gui.sh` to `b/git-gui.sh` can be applied to the file in the working tree `modules/git-gui/git-gui.sh` by running `git apply --directory=modules/git-gui`.

--unsafe-paths

By default, a patch that affects outside the working area (either a Git controlled working tree, or the current working directory when "git apply" is used as a replacement of GNU patch) is rejected as a mistake (or a mischief).

When `git apply` is used as a "better GNU patch", the user can pass the `--unsafe-paths` option to override this safety check. This option has no effect when `--index` or `--cached` is in use.

Configuration

apply.ignoreWhitespace

Set to *change* if you want changes in whitespace to be ignored by default. Set to one of: no, none, never, false if you want changes in whitespace to be significant.

apply.whitespace

When no `--whitespace` flag is given from the command line, this configuration item is used as the default.

Submodules

If the patch contains any changes to submodules then *git apply* treats these changes as follows.

If `--index` is specified (explicitly or implicitly), then the submodule commits must match the index exactly for the patch to apply. If any of the submodules are checked-out, then these check-outs are completely ignored, i.e., they are not required to be up-to-date or clean and they are not updated.

If `--index` is not specified, then the submodule commits in the patch are ignored and only the absence or presence of the corresponding subdirectory is checked and (if possible) updated.

SEE ALSO

git-am[1].

GIT

Part of the git[1] suite

format-patch

NAME

git-format-patch - Prepare patches for e-mail submission

SYNOPSIS

```
git format-patch [-k] [(-o|--output-directory) <dir> | --stdout]
           [--no-thread | --thread[=<style>]]
           [(--attach|--inline)[=<boundary>] | --no-attach]
           [-s | --signoff]
           [--signature=<signature> | --no-signature]
           [--signature-file=<file>]
           [-n | --numbered | -N | --no-numbered]
           [--start-number <n>] [--numbered-files]
           [--in-reply-to=Message-Id] [--suffix=.<sfx>]
           [--ignore-if-in-upstream]
           [--subject-prefix=Subject-Prefix] [(--reroll-count|-v) <n>]
           [--to=<email>] [--cc=<email>]
           [--[no-]cover-letter] [--quiet] [--notes[=<ref>]]
           [<common diff options>]
           [ <since> | <revision range> ]
```

DESCRIPTION

Prepare each commit with its patch in one file per commit, formatted to resemble UNIX mailbox format. The output of this command is convenient for e-mail submission or for use with *git am*.

There are two ways to specify which commits to operate on.

1. A single commit, <since>, specifies that the commits leading to the tip of the current branch that are not in the history that leads to the <since> to be output.
2. Generic <revision range> expression (see "SPECIFYING REVISIONS" section in gitrevisions[7]) means the commits in the specified range.

The first rule takes precedence in the case of a single <commit>. To apply the second rule, i.e., format everything since the beginning of history up until <commit>, use the *--root* option: `git format-patch --root <commit>`. If you want to format only <commit> itself, you can do this with `git format-patch -1 <commit>`.

By default, each output file is numbered sequentially from 1, and uses the first line of the commit message (massaged for pathname safety) as the filename. With the `--numbered-files` option, the output file names will only be numbers, without the first line of the commit appended. The names of the output files are printed to standard output, unless the `--stdout` option is specified.

If `-o` is specified, output files are created in <dir>. Otherwise they are created in the current working directory. The default path can be set with the *format.outputDirectory* configuration option. The `-o` option takes precedence over `format.outputDirectory`. To store patches in the current working directory even when `format.outputDirectory` points elsewhere, use `-o .`.

By default, the subject of a single patch is "[PATCH] " followed by the concatenation of lines from the commit message up to the first blank line (see the DISCUSSION section of git-commit[1]).

When multiple patches are output, the subject prefix will instead be "[PATCH n/m] ". To force 1/1 to be added for a single patch, use `-n`. To omit patch numbers from the subject, use `-N`.

If given `--thread`, `git-format-patch` will generate `In-Reply-To` and `References` headers to make the second and subsequent patch mails appear as replies to the first mail; this also generates a `Message-Id` header to reference.

OPTIONS

-p

--no-stat

Generate plain patches without any diffstats.

-s

--no-patch

Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>

Generate diffs with <n> lines of context instead of the usual three.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

`default` , `myers`

The basic greedy diff algorithm. Currently, this is the default.

`minimal`

Spend extra time to make sure the smallest possible diff is produced.

`patience`

Use "patience diff" algorithm when generating patches.

`histogram`

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured diff.algorithm variable to a non-default value and want to use the default one, then you have to use `--diff-algorithm=default` option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by `<width>` . The width of the filename part can be limited by giving another width `<name-width>` after a comma. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by setting `diff.statGraphWidth=<width>` (does not affect `git format-patch`). By giving a third parameter `<count>` , you can limit the output to the first `<count>` lines, followed by `...` if there are more.

These parameters can also be set individually with `--stat-width=<width>` , `--stat-name-width=<name-width>` and `--stat-count=<count>` .

--numstat

Similar to `--stat`, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two `-` instead of saying `0 0`.

--shortstat

Output only the last line of the `--stat` format containing total number of modified files, as well as number of added and deleted lines.

--dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of `--dirstat` can be customized by passing it a comma separated list of parameters. The defaults are controlled by the `diff.dirstat` configuration variable (see git-config[1]). The following parameters are available:

`changes`

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

`lines`

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive `--dirstat` behavior than the `changes` behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other `--*stat` options.

`files`

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest `--dirstat` behavior, since it does not have to look at the file contents at all.

`cumulative`

Count changes in a child directory for the parent directory as well. Note that when using `cumulative`, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the `noncumulative` parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: `--dirstat=files,10,cumulative`.

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to `--full-index`, output a binary diff that can be applied with `git-apply`.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. Non default number of digits can be specified with `--abbrev=<n>`.

-B[<n>][/<m>]

--break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number `m` controls this aspect of the -B option (defaults to 60%). `-B/70%` specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number `n` controls this aspect of the -B option (defaults to 50%). `-B20%` specifies that a

change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>]

--find-renames[=<n>]

Detect renames. If `n` is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, `-M90%` means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a `%` sign, the number is to be read as a fraction, with a decimal point before it. I.e., `-M5` becomes 0.5, and is thus the same as `-M50%`. Similarly, `-M05` is the same as `-M5%`. To limit detection to exact renames, use `-M100%`. The default similarity index is 50%.

-C[<n>]

--find-copies[=<n>]

Detect copies as well as renames. See also `--find-copies-harder`. If `n` is specified, it has the same meaning as for `-M<n>`.

--find-copies-harder

For performance reasons, by default, `-C` option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one `-C` option has the same effect.

-D

--irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and `/dev/null`. The resulting patch is not meant to be applied with `patch` or `git apply`; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lack enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with `-B`, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The `-M` and `-C` options require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

-O<orderfile>

Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config[1]). To cancel `diff.orderFile`, use `-O/dev/null`.

-a

--text

Treat all files as text.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b

--ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w

--ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other.

-W

--function-context

Show whole surrounding functions of changes.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with gitattributes[5], you need to use this option with git-log[1] and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv

--no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See gitattributes[5] for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for git-diff[1] and git-log[1], but not for git-format-patch[1] or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in git-config[1] or gitmodules[5]. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

For more detailed explanation on these common options, see also gitdiffcore[7].

-<n>

Prepare patches from the topmost <n> commits.

-o <dir>

--output-directory <dir>

Use <dir> to store the resulting files, instead of the current working directory.

-n

--numbered

Name output in *[PATCH n/m]* format, even with a single patch.

-N

--no-numbered

Name output in *[PATCH]* format.

--start-number <n>

Start numbering the patches at <n> instead of 1.

--numbered-files

Output file names will be a simple number sequence without the default first line of the commit appended.

-k

--keep-subject

Do not strip/add *[PATCH]* from the first line of the commit log message.

-s

--signoff

Add `Signed-off-by:` line to the commit message, using the committer identity of yourself. See the signoff option in git-commit[1] for more information.

--stdout

Print all commits to the standard output in mbox format, instead of creating a file for each one.

--attach[=<boundary>]

Create multipart/mixed attachment, the first part of which is the commit message and the patch itself in the second part, with `Content-Disposition: attachment`.

--no-attach

Disable the creation of an attachment, overriding the configuration setting.

--inline[=<boundary>]

Create multipart/mixed attachment, the first part of which is the commit message and the patch itself in the second part, with `Content-Disposition: inline`.

--thread[=<style>]

--no-thread

Controls addition of `In-Reply-To` and `References` headers to make the second and subsequent mails appear as replies to the first. Also controls generation of the `Message-Id` header to reference.

The optional <style> argument can be either `shallow` or `deep`. *shallow* threading makes every mail a reply to the head of the series, where the head is chosen from the cover letter, the `--in-reply-to`, and the first patch mail, in this order. *deep* threading makes every mail a reply to the previous one.

The default is `--no-thread`, unless the *format.thread* configuration is set. If `--thread` is specified without a style, it defaults to the style specified by *format.thread* if any, or else `shallow`.

Beware that the default for *git send-email* is to thread emails itself. If you want `git format-patch` to take care of threading, you will want to ensure that threading is disabled for `git send-email`.

--in-reply-to=Message-Id

Make the first mail (or all the mails with `--no-thread`) appear as a reply to the given Message-Id, which avoids breaking threads to provide a new patch series.

--ignore-if-in-upstream

Do not include a patch that matches a commit in <until>..<since>. This will examine all patches reachable from <since> but not from <until> and compare them with the patches being generated, and any patch that matches is ignored.

--subject-prefix=<Subject-Prefix>

Instead of the standard *[PATCH]* prefix in the subject line, instead use *[<Subject-Prefix>]*. This allows for useful naming of a patch series, and can be combined with the `--numbered` option.

-v <n>

--reroll-count=<n>

Mark the series as the <n>-th iteration of the topic. The output filenames have `v<n>` prepended to them, and the subject prefix ("PATCH" by default, but configurable via the `--subject-prefix` option) has `v<n>` appended to it. E.g. `--reroll-count=4` may produce `v4-0001-add-makefile.patch` file that has "Subject: [PATCH v4 1/20] Add makefile" in it.

--to=<email>

Add a `To:` header to the email headers. This is in addition to any configured headers, and may be used multiple times. The negated form `--no-to` discards all `To:` headers added so far (from config or command line).

--cc=<email>

Add a `Cc:` header to the email headers. This is in addition to any configured headers, and may be used multiple times. The negated form `--no-cc` discards all `Cc:` headers added so far (from config or command line).

--from

--from=<ident>

Use `ident` in the `From:` header of each commit email. If the author ident of the commit is not textually identical to the provided `ident`, place a `From:` header in the body of the message with the original author. If no `ident` is given, use the committer ident.

Note that this option is only useful if you are actually sending the emails and want to identify yourself as the sender, but retain the original author (and `git am` will correctly pick up the in-body header). Note also that `git send-email` already handles this transformation for you, and this option should not be used if you are feeding the result to `git send-email`.

--add-header=<header>

Add an arbitrary header to the email headers. This is in addition to any configured headers, and may be used multiple times. For example, `--add-header="Organization: git-foo"`. The negated form `--no-add-header` discards **all** (`To:`, `Cc:`, and custom) headers added so far from config or command line.

--[no-]cover-letter

In addition to the patches, generate a cover letter file containing the branch description, shortlog and the overall diffstat. You can fill in a description in the file before sending it out.

--notes[=<ref>]

Append the notes (see git-notes[1]) for the commit after the three-dash line.

The expected use case of this is to write supporting explanation for the commit that does not belong to the commit log message proper, and include it with the patch submission. While one can simply write these explanations after `format-patch` has run but before sending, keeping them as Git notes allows them to be maintained between versions of the patch series (but see the discussion of the `notes.rewrite` configuration options in git-notes[1] to use this workflow).

--[no]-signature=<signature>

Add a signature to each message produced. Per RFC 3676 the signature is separated from the body by a line with '-- ' on it. If the signature option is omitted the signature defaults to the Git version number.

--signature-file=<file>

Works just like --signature except the signature is read from a file.

--suffix=.<sfx>

Instead of using `.patch` as the suffix for generated filenames, use specified suffix. A common alternative is `--suffix=.txt`. Leaving this empty will remove the `.patch` suffix.

Note that the leading character does not have to be a dot; for example, you can use `--suffix=-patch` to get `0001-description-of-my-change-patch`.

-q

--quiet

Do not print the names of the generated files to standard output.

--no-binary

Do not output contents of changes in binary files, instead display a notice that those files changed. Patches generated using this option cannot be applied properly, but they are still useful for code review.

--zero-commit

Output an all-zero hash in each patch’s From header instead of the hash of the commit.

--root

Treat the revision argument as a <revision range>, even if it is just a single commit (that would normally be treated as a <since>). Note that root commits included in the specified range are always formatted as creation patches, independently of this flag.

CONFIGURATION

You can specify extra mail header lines to be added to each message, defaults for the subject prefix and file suffix, number patches when outputting more than one patch, add "To" or "Cc:" headers, configure attachments, and sign off patches with configuration variables.

```
[format]
    headers = "Organization: git-foo\n"
    subjectPrefix = CHANGE
    suffix = .txt
    numbered = auto
    to = <email>
    cc = <email>
    attach [ = mime-boundary-string ]
    signOff = true
    coverletter = auto
```

DISCUSSION

The patch produced by *git format-patch* is in UNIX mailbox format, with a fixed "magic" time stamp to indicate that the file is output from format-patch rather than a real mailbox, like so:

```
From 8f72bad1baf19a53459661343e21d6491c3908d3 Mon Sep 17 00:00:00 2001
From: Tony Luck <tony.luck@intel.com>
Date: Tue, 13 Jul 2010 11:42:54 -0700
Subject: [PATCH] =?UTF-8?q?[IA64]=20Put=20ia64=20config=20files=20on=20the=20?=
 =?UTF-8?q?Uwe=20Kleine-K=C3=B6nig=20diet?=
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 8bit

arch/arm config files were slimmed down using a python script
(See commit c2330e286f68f1c408b4aa6515ba49d57f05beae comment)

Do the same for ia64 so we can have sleek & trim looking
...
```

Typically it will be placed in a MUA's drafts folder, edited to add timely commentary that should not go in the changelog after the three dashes, and then sent as a message whose body, in our example, starts with "arch/arm config files were…". On the receiving end, readers can save interesting patches in a UNIX mailbox and apply them with git-am[1].

When a patch is part of an ongoing discussion, the patch generated by *git format-patch* can be tweaked to take advantage of the *git am --scissors* feature. After your response to the discussion comes a line that consists solely of " `-- >8 --` " (scissors and perforation), followed by the patch with unnecessary header fields removed:

```
...
> So we should do such-and-such.

Makes sense to me.  How about this patch?

-- >8 --
Subject: [IA64] Put ia64 config files on the Uwe Kleine-K��nig diet

arch/arm config files were slimmed down using a python script
...
```

When sending a patch this way, most often you are sending your own patch, so in addition to the " `From $SHA1 $magic_timestamp` " marker you should omit `From:` and `Date:` lines from the patch file. The patch title is likely to be different from the subject of the discussion the patch is in response to, so it is likely that you would want to keep the Subject: line, like the example above.

Checking for patch corruption

Many mailers if not set up properly will corrupt whitespace. Here are two common types of corruption:

- Empty context lines that do not have *any* whitespace.
- Non-empty context lines that have one extra whitespace at the beginning.

One way to test if your MUA is set up correctly is:

- Send the patch to yourself, exactly the way you would, except with To: and Cc: lines that do not contain the list and maintainer address.
- Save that patch to a file in UNIX mailbox format. Call it a.patch, say.
- Apply it:

  ```
  $ git fetch &lt;project&gt; master:test-apply
  $ git checkout test-apply
  $ git reset --hard
  $ git am a.patch
  ```

If it does not apply correctly, there can be various reasons.

- The patch itself does not apply cleanly. That is *bad* but does not have much to do with your MUA. You might want to rebase the patch with git-rebase[1] before regenerating it in this case.
- The MUA corrupted your patch; "am" would complain that the patch does not apply. Look in the .git/rebase-apply/ subdirectory and see what *patch* file contains and check for the common corruption patterns mentioned above.
- While at it, check the *info* and *final-commit* files as well. If what is in *final-commit* is not exactly what you would want to see in the commit log message, it is very likely that the receiver would end up hand editing the log message when applying your patch. Things like "Hi, this is my first patch.\n" in the patch e-mail should come after the three-dash line that signals the end of the commit message.

MUA-SPECIFIC HINTS

Here are some hints on how to successfully submit patches inline using various mailers.

GMail

GMail does not have any way to turn off line wrapping in the web interface, so it will mangle any emails that you send. You can however use "git send-email" and send your patches through the GMail SMTP server, or use any IMAP email client to connect to the google IMAP server and forward the emails through that.

For hints on using *git send-email* to send your patches through the GMail SMTP server, see the EXAMPLE section of git-send-email[1].

For hints on submission using the IMAP interface, see the EXAMPLE section of git-imap-send[1].

Thunderbird

By default, Thunderbird will both wrap emails as well as flag them as being *format=flowed*, both of which will make the resulting email unusable by Git.

There are three different approaches: use an add-on to turn off line wraps, configure Thunderbird to not mangle patches, or use an external editor to keep Thunderbird from mangling the patches.

Approach #1 (add-on)

Install the Toggle Word Wrap add-on that is available from https://addons.mozilla.org/thunderbird/addon/toggle-word-wrap/ It adds a menu entry "Enable Word Wrap" in the composer's "Options" menu that you can tick off. Now you can compose the message as you otherwise do (cut + paste, *git format-patch* | *git imap-send*, etc), but you have to insert line breaks manually in any text that you type.

Approach #2 (configuration)

Three steps:

1. Configure your mail server composition as plain text: Edit…Account Settings…Composition & Addressing, uncheck "Compose Messages in HTML".

2. Configure your general composition window to not wrap.

   In Thunderbird 2: Edit..Preferences..Composition, wrap plain text messages at 0

In Thunderbird 3: Edit..Preferences..Advanced..Config Editor. Search for "mail.wrap_long_lines". Toggle it to make sure it is set to `false`. Also, search for "mailnews.wraplength" and set the value to 0.

3. Disable the use of format=flowed: Edit..Preferences..Advanced..Config Editor. Search for "mailnews.send_plaintext_flowed". Toggle it to make sure it is set to `false`.

After that is done, you should be able to compose email as you otherwise would (cut + paste, *git format-patch* | *git imap-send*, etc), and the patches will not be mangled.

Approach #3 (external editor)

The following Thunderbird extensions are needed: AboutConfig from http://aboutconfig.mozdev.org/ and External Editor from http://globs.org/articles.php?lng=en&pg=8

1. Prepare the patch as a text file using your method of choice.
2. Before opening a compose window, use Edit→Account Settings to uncheck the "Compose messages in HTML format" setting in the "Composition & Addressing" panel of the account to be used to send the patch.
3. In the main Thunderbird window, *before* you open the compose window for the patch, use Tools→about:config to set the following to the indicated values:

```
    mailnews.send_plaintext_flowed  =&gt; false
    mailnews.wraplength             =&gt; 0
```

4. Open a compose window and click the external editor icon.
5. In the external editor window, read in the patch file and exit the editor normally.

Side note: it may be possible to do step 2 with about:config and the following settings but no one’s tried yet.

```
    mail.html_compose                       => false
    mail.identity.default.compose_html      => false
    mail.identity.id?.compose_html          => false
```

There is a script in contrib/thunderbird-patch-inline which can help you include patches with Thunderbird in an easy way. To use it, do the steps above and then use the script as the external editor.

KMail

This should help you to submit patches inline using KMail.

1. Prepare the patch as a text file.
2. Click on New Mail.
3. Go under "Options" in the Composer window and be sure that "Word wrap" is not set.
4. Use Message → Insert file… and insert the patch.
5. Back in the compose window: add whatever other text you wish to the message, complete the addressing and subject fields, and press send.

EXAMPLES

- Extract commits between revisions R1 and R2, and apply them on top of the current branch using *git am* to cherry-pick them:

  ```
  $ git format-patch -k --stdout R1..R2 | git am -3 -k
  ```

- Extract all commits which are in the current branch but not in the origin branch:

  ```
  $ git format-patch origin
  ```

  For each commit a separate file is created in the current directory.

- Extract all commits that lead to *origin* since the inception of the project:

  ```
  $ git format-patch --root origin
  ```

- The same as the previous one:

  ```
  $ git format-patch -M -B origin
  ```

  Additionally, it detects and handles renames and complete rewrites intelligently to produce a renaming patch. A renaming patch reduces the amount of text output, and generally makes it easier to review. Note that non-Git "patch" programs won’t understand renaming patches, so use it only when you know the recipient uses Git to apply your patch.

- Extract three topmost commits from the current branch and format them as e-mailable patches:

  ```
  $ git format-patch -3
  ```

SEE ALSO

git-am[1], git-send-email[1]

GIT

Part of the git[1] suite

send-email

NAME

git-send-email - Send a collection of patches as emails

SYNOPSIS

```
git send-email [options] <file|directory|rev-list options>…
git send-email --dump-aliases
```

DESCRIPTION

Takes the patches given on the command line and emails them out. Patches can be specified as files, directories (which will send all files in the directory), or directly as a revision list. In the last case, any format accepted by git-format-patch[1] can be passed to git send-email.

The header of the email is configurable via command-line options. If not specified on the command line, the user will be prompted with a ReadLine enabled interface to provide the necessary information.

There are two formats accepted for patch files:

1. mbox format files

   This is what git-format-patch[1] generates. Most headers and MIME formatting are ignored.

2. The original format used by Greg Kroah-Hartman's *send_lots_of_email.pl* script

   This format expects the first line of the file to contain the "Cc:" value and the "Subject:" of the message as the second line.

OPTIONS

Composing

--annotate

Review and edit each patch you're about to send. Default is the value of *sendemail.annotate*. See the CONFIGURATION section for *sendemail.multiEdit*.

--bcc=<address>,…

Specify a "Bcc:" value for each email. Default is the value of *sendemail.bcc*.

This option may be specified multiple times.

--cc=<address>,…

Specify a starting "Cc:" value for each email. Default is the value of *sendemail.cc*.

This option may be specified multiple times.

--compose

Invoke a text editor (see GIT_EDITOR in git-var[1]) to edit an introductory message for the patch series.

When *--compose* is used, git send-email will use the From, Subject, and In-Reply-To headers specified in the message. If the body of the message (what you type after the headers and a blank line) only contains blank (or Git: prefixed) lines, the summary won't be sent, but From, Subject, and In-Reply-To headers will be used unless they are removed.

Missing From or In-Reply-To headers will be prompted for.

See the CONFIGURATION section for *sendemail.multiEdit*.

--from=<address>

Specify the sender of the emails. If not specified on the command line, the value of the *sendemail.from* configuration option is used. If neither the command-line option nor *sendemail.from* are set, then the user will be prompted for the value. The default for the prompt will be the value of GIT_AUTHOR_IDENT, or GIT_COMMITTER_IDENT if that is not set, as returned by "git var -l".

--in-reply-to=<identifier>

Make the first mail (or all the mails with `--no-thread`) appear as a reply to the given Message-Id, which avoids breaking threads to provide a new patch series. The second and subsequent emails will be sent as replies according to the `--[no]-chain-reply-to` setting.

So for example when `--thread` and `--no-chain-reply-to` are specified, the second and subsequent patches will be replies to the first one like in the illustration below where `[PATCH v2 0/3]` is in reply to `[PATCH 0/2]`:

```
[PATCH 0/2] Here is what I did...
  [PATCH 1/2] Clean up and tests
  [PATCH 2/2] Implementation
  [PATCH v2 0/3] Here is a reroll
    [PATCH v2 1/3] Clean up
    [PATCH v2 2/3] New tests
    [PATCH v2 3/3] Implementation
```

Only necessary if --compose is also set. If --compose is not set, this will be prompted for.

--subject=<string>

Specify the initial subject of the email thread. Only necessary if --compose is also set. If --compose is not set, this will be prompted for.

--to=<address>,…

Specify the primary recipient of the emails generated. Generally, this will be the upstream maintainer of the project involved. Default is the value of the *sendemail.to* configuration value; if that is unspecified, and --to-cmd is not specified, this will be prompted for.

This option may be specified multiple times.

--8bit-encoding=<encoding>

When encountering a non-ASCII message or subject that does not declare its encoding, add headers/quoting to indicate it is encoded in <encoding>. Default is the value of the *sendemail.assume8bitEncoding*; if that is unspecified, this will be prompted for if any non-ASCII files are encountered.

Note that no attempts whatsoever are made to validate the encoding.

--compose-encoding=<encoding>

Specify encoding of compose message. Default is the value of the *sendemail.composeencoding*; if that is unspecified, UTF-8 is assumed.

--transfer-encoding=(7bit|8bit|quoted-printable|base64)

Specify the transfer encoding to be used to send the message over SMTP. 7bit will fail upon encountering a non-ASCII message. quoted-printable can be useful when the repository contains files that contain carriage returns, but makes the raw patch email file (as saved from a MUA) much harder to inspect manually. base64 is even more fool proof, but also even more opaque. Default is the value of the *sendemail.transferEncoding* configuration value; if that is unspecified, git will use 8bit and not add a Content-Transfer-Encoding header.

--xmailer

--no-xmailer

Add (or prevent adding) the "X-Mailer:" header. By default, the header is added, but it can be turned off by setting the `sendemail.xmailer` configuration variable to `false`.

Sending

--envelope-sender=<address>

Specify the envelope sender used to send the emails. This is useful if your default address is not the address that is subscribed to a list. In order to use the *From* address, set the value to "auto". If you use the sendmail binary, you must have suitable privileges for the -f parameter. Default is the value of the *sendemail.envelopeSender* configuration variable; if that is unspecified, choosing the envelope sender is left to your MTA.

--smtp-encryption=<encryption>

Specify the encryption to use, either *ssl* or *tls*. Any other value reverts to plain SMTP. Default is the value of *sendemail.smtpEncryption*.

--smtp-domain=<FQDN>

Specifies the Fully Qualified Domain Name (FQDN) used in the HELO/EHLO command to the SMTP server. Some servers require the FQDN to match your IP address. If not set, git send-email attempts to determine your FQDN automatically. Default is the value of *sendemail.smtpDomain*.

--smtp-auth=<mechanisms>

Whitespace-separated list of allowed SMTP-AUTH mechanisms. This setting forces using only the listed mechanisms. Example:

```
$ git send-email --smtp-auth="PLAIN LOGIN GSSAPI" ...
```

If at least one of the specified mechanisms matches the ones advertised by the SMTP server and if it is supported by the utilized SASL library, the mechanism is used for authentication. If neither *sendemail.smtpAuth* nor *--smtp-auth* is specified, all mechanisms supported by the SASL library can be used.

--smtp-pass[=<password>]

Password for SMTP-AUTH. The argument is optional: If no argument is specified, then the empty string is used as the password. Default is the value of *sendemail.smtpPass*, however *--smtp-pass* always overrides this value.

Furthermore, passwords need not be specified in configuration files or on the command line. If a username has been specified (with *--smtp-user* or a *sendemail.smtpUser*), but no password has been specified (with *--smtp-pass* or *sendemail.smtpPass*), then a password is obtained using *git-credential*.

--smtp-server=<host>

If set, specifies the outgoing SMTP server to use (e.g. `smtp.example.com` or a raw IP address). Alternatively it can specify a full pathname of a sendmail-like program instead; the program must support the `-i` option. Default value can be specified by the *sendemail.smtpServer* configuration option; the built-in default is `/usr/sbin/sendmail` or `/usr/lib/sendmail` if such program is available, or `localhost` otherwise.

--smtp-server-port=<port>

Specifies a port different from the default port (SMTP servers typically listen to smtp port 25, but may also listen to submission port 587, or the common SSL smtp port 465); symbolic port names (e.g. "submission" instead of 587) are also accepted. The port can also be set with the *sendemail.smtpServerPort* configuration variable.

--smtp-server-option=<option>

If set, specifies the outgoing SMTP server option to use. Default value can be specified by the *sendemail.smtpServerOption* configuration option.

The --smtp-server-option option must be repeated for each option you want to pass to the server. Likewise, different lines in the configuration files must be used for each option.

--smtp-ssl

Legacy alias for *--smtp-encryption ssl*.

--smtp-ssl-cert-path

Path to a store of trusted CA certificates for SMTP SSL/TLS certificate validation (either a directory that has been processed by *c_rehash*, or a single file containing one or more PEM format certificates concatenated together: see verify(1) -CAfile and -CApath for more information on these). Set it to an empty string to disable certificate verification. Defaults to the value of the *sendemail.smtpsslcertpath* configuration variable, if set, or the backing SSL library's compiled-in default otherwise (which should be the best choice on most platforms).

--smtp-user=<user>

Username for SMTP-AUTH. Default is the value of *sendemail.smtpUser*; if a username is not specified (with *--smtp-user* or *sendemail.smtpUser*), then authentication is not attempted.

--smtp-debug=0|1

Enable (1) or disable (0) debug output. If enabled, SMTP commands and replies will be printed. Useful to debug TLS connection and authentication problems.

Automating

--to-cmd=<command>

Specify a command to execute once per patch file which should generate patch file specific "To:" entries. Output of this command must be single email address per line. Default is the value of *sendemail.tocmd* configuration value.

--cc-cmd=<command>

Specify a command to execute once per patch file which should generate patch file specific "Cc:" entries. Output of this command must be single email address per line. Default is the value of *sendemail.ccCmd* configuration value.

--[no-]chain-reply-to

If this is set, each email will be sent as a reply to the previous email sent. If disabled with "--no-chain-reply-to", all emails after the first will be sent as replies to the first email sent. When using this, it is recommended that the first file given be an overview of the entire patch series. Disabled by default, but the *sendemail.chainReplyTo* configuration variable can be used to enable it.

--identity=<identity>

A configuration identity. When given, causes values in the *sendemail.<identity>* subsection to take precedence over values in the *sendemail* section. The default identity is the value of *sendemail.identity*.

--[no-]signed-off-by-cc

If this is set, add emails found in Signed-off-by: or Cc: lines to the cc list. Default is the value of *sendemail.signedoffbycc* configuration value; if that is unspecified, default to --signed-off-by-cc.

--[no-]cc-cover

If this is set, emails found in Cc: headers in the first patch of the series (typically the cover letter) are added to the cc list for each email set. Default is the value of *sendemail.cccover* configuration value; if that is unspecified, default to --no-cc-cover.

--[no-]to-cover

If this is set, emails found in To: headers in the first patch of the series (typically the cover letter) are added to the to list for each email set. Default is the value of *sendemail.tocover* configuration value; if that is unspecified, default to --no-to-cover.

--suppress-cc=<category>

Specify an additional category of recipients to suppress the auto-cc of:

- *author* will avoid including the patch author
- *self* will avoid including the sender
- *cc* will avoid including anyone mentioned in Cc lines in the patch header except for self (use *self* for that).
- *bodycc* will avoid including anyone mentioned in Cc lines in the patch body (commit message) except for self (use *self* for that).
- *sob* will avoid including anyone mentioned in Signed-off-by lines except for self (use *self* for that).
- *cccmd* will avoid running the --cc-cmd.
- *body* is equivalent to *sob* + *bodycc*
- *all* will suppress all auto cc values.

Default is the value of *sendemail.suppresscc* configuration value; if that is unspecified, default to *self* if --suppress-from is specified, as well as *body* if --no-signed-off-cc is specified.

--[no-]suppress-from

If this is set, do not add the From: address to the cc: list. Default is the value of *sendemail.suppressFrom* configuration value; if that is unspecified, default to --no-suppress-from.

--[no-]thread

If this is set, the In-Reply-To and References headers will be added to each email sent. Whether each mail refers to the previous email (`deep` threading per *git format-patch* wording) or to the first email (`shallow` threading) is governed by "--[no-]chain-reply-to".

If disabled with "--no-thread", those headers will not be added (unless specified with --in-reply-to). Default is the value of the *sendemail.thread* configuration value; if that is unspecified, default to --thread.

It is up to the user to ensure that no In-Reply-To header already exists when *git send-email* is asked to add it (especially note that *git format-patch* can be configured to do the threading itself). Failure to do so may not produce the expected result in the recipient's MUA.

Administering

--confirm=<mode>

Confirm just before sending:

- *always* will always confirm before sending
- *never* will never confirm before sending
- *cc* will confirm before sending when send-email has automatically added addresses from the patch to the Cc list
- *compose* will confirm before sending the first message when using --compose.
- *auto* is equivalent to *cc* + *compose*

Default is the value of *sendemail.confirm* configuration value; if that is unspecified, default to *auto* unless any of the suppress options have been specified, in which case default to *compose*.

--dry-run

Do everything except actually send the emails.

--[no-]format-patch

When an argument may be understood either as a reference or as a file name, choose to understand it as a format-patch argument (*--format-patch*) or as a file name (*--no-format-patch*). By default, when such a conflict occurs, git send-email will fail.

--quiet

Make git-send-email less verbose. One line per email should be all that is output.

--[no-]validate

Perform sanity checks on patches. Currently, validation means the following:

- Warn of patches that contain lines longer than 998 characters; this is due to SMTP limits as described by http://www.ietf.org/rfc/rfc2821.txt.

Default is the value of *sendemail.validate*; if this is not set, default to *--validate*.

--force

Send emails even if safety checks would prevent it.

Information

--dump-aliases

Instead of the normal operation, dump the shorthand alias names from the configured alias file(s), one per line in alphabetical order. Note, this only includes the alias name and not its expanded email addresses. See *sendemail.aliasesfile* for more information about aliases.

CONFIGURATION

sendemail.aliasesFile

To avoid typing long email addresses, point this to one or more email aliases files. You must also supply *sendemail.aliasFileType*.

sendemail.aliasFileType

Format of the file(s) specified in sendemail.aliasesFile. Must be one of *mutt*, *mailrc*, *pine*, *elm*, or *gnus*, or *sendmail*.

What an alias file in each format looks like can be found in the documentation of the email program of the same name. The differences and limitations from the standard formats are described below:

sendmail

- Quoted aliases and quoted addresses are not supported: lines that contain a `"` symbol are ignored.
- Redirection to a file (`/path/name`) or pipe (`|command`) is not supported.
- File inclusion (`:include: /path/name`) is not supported.
- Warnings are printed on the standard error output for any explicitly unsupported constructs, and any other lines that are not recognized by the parser.

sendemail.multiEdit

If true (default), a single editor instance will be spawned to edit files you have to edit (patches when *--annotate* is used, and the summary when *--compose* is used). If false, files will be edited one after the other, spawning a new editor each time.

sendemail.confirm

Sets the default for whether to confirm before sending. Must be one of *always*, *never*, *cc*, *compose*, or *auto*. See *--confirm* in the previous section for the meaning of these values.

EXAMPLE

Use gmail as the smtp server

To use *git send-email* to send your patches through the GMail SMTP server, edit ~/.gitconfig to specify your account settings:

```
[sendemail]
    smtpEncryption = tls
    smtpServer = smtp.gmail.com
    smtpUser = yourname@gmail.com
    smtpServerPort = 587
```

Once your commits are ready to be sent to the mailing list, run the following commands:

```
$ git format-patch --cover-letter -M origin/master -o outgoing/
$ edit outgoing/0000-*
$ git send-email outgoing/*
```

Note: the following perl modules are required Net::SMTP::SSL, MIME::Base64 and Authen::SASL

SEE ALSO

git-format-patch[1], git-imap-send[1], mbox(5)

GIT

Part of the git[1] suite

request-pull

NAME

git-request-pull - Generates a summary of pending changes

SYNOPSIS

```
git request-pull [-p] <start> <url> [<end>]
```

DESCRIPTION

Generate a request asking your upstream project to pull changes into their tree. The request, printed to the standard output, begins with the branch description, summarizes the changes and indicates from where they can be pulled.

The upstream project is expected to have the commit named by `<start>` and the output asks it to integrate the changes you made since that commit, up to the commit named by `<end>`, by visiting the repository named by `<url>`.

OPTIONS

-p

Include patch text in the output.

<start>

Commit to start at. This names a commit that is already in the upstream history.

<url>

The repository URL to be pulled from.

<end>

Commit to end at (defaults to HEAD). This names the commit at the tip of the history you are asking to be pulled.

When the repository named by `<url>` has the commit at a tip of a ref that is different from the ref you have locally, you can use the `<local>:<remote>` syntax, to have its local name, a colon `:`, and its remote name.

EXAMPLE

Imagine that you built your work on your `master` branch on top of the `v1.0` release, and want it to be integrated to the project. First you push that change to your public repository for others to see:

```
git push https://git.ko.xz/project master
```

Then, you run this command:

```
git request-pull v1.0 https://git.ko.xz/project master
```

which will produce a request to the upstream, summarizing the changes between the `v1.0` release and your `master`, to pull it from your public repository.

If you pushed your change to a branch whose name is different from the one you have locally, e.g.

```
git push https://git.ko.xz/project master:for-linus
```

then you can ask that to be pulled with

```
git request-pull v1.0 https://git.ko.xz/project master:for-linus
```

GIT

Part of the git[1] suite

External Systems

svn

NAME

git-svn - Bidirectional operation between a Subversion repository and Git

SYNOPSIS

```
git svn <command> [options] [arguments]
```

DESCRIPTION

git svn is a simple conduit for changesets between Subversion and Git. It provides a bidirectional flow of changes between a Subversion and a Git repository.

git svn can track a standard Subversion repository, following the common "trunk/branches/tags" layout, with the --stdlayout option. It can also follow branches and tags in any layout with the -T/-t/-b options (see options to *init* below, and also the *clone* command).

Once tracking a Subversion repository (with any of the above methods), the Git repository can be updated from Subversion by the *fetch* command and Subversion updated from Git by the *dcommit* command.

COMMANDS

init

Initializes an empty Git repository with additional metadata directories for *git svn*. The Subversion URL may be specified as a command-line argument, or as full URL arguments to -T/-t/-b. Optionally, the target directory to operate on can be specified as a second argument. Normally this command initializes the current directory.

-T<trunk_subdir>

--trunk=<trunk_subdir>

-t<tags_subdir>

--tags=<tags_subdir>

-b<branches_subdir>

--branches=<branches_subdir>

-s

--stdlayout

These are optional command-line options for init. Each of these flags can point to a relative repository path (--tags=project/tags) or a full url (--tags=https://foo.org/project/tags). You can specify more than one --tags and/or --branches options, in case your Subversion repository places tags or branches under multiple paths. The option --stdlayout is a shorthand way of setting trunk,tags,branches as the relative paths, which is the Subversion default. If any of the other options are given as well, they take precedence.

--no-metadata

Set the *noMetadata* option in the [svn-remote] config. This option is not recommended, please read the *svn.noMetadata* section of this manpage before using this option.

--use-svm-props

Set the *useSvmProps* option in the [svn-remote] config.

--use-svnsync-props

Set the *useSvnsyncProps* option in the [svn-remote] config.

--rewrite-root=<URL>

Set the *rewriteRoot* option in the [svn-remote] config.

--rewrite-uuid=<UUID>

Set the *rewriteUUID* option in the [svn-remote] config.

--username=<user>

For transports that SVN handles authentication for (http, https, and plain svn), specify the username. For other transports (e.g. `svn+ssh://`), you must include the username in the URL, e.g. `svn+ssh://foo@svn.bar.com/project`

--prefix=<prefix>

This allows one to specify a prefix which is prepended to the names of remotes if trunk/branches/tags are specified. The prefix does not automatically include a trailing slash, so be sure you include one in the argument if that is what you want. If --branches/-b is

specified, the prefix must include a trailing slash. Setting a prefix (with a trailing slash) is strongly encouraged in any case, as your SVN-tracking refs will then be located at "refs/remotes/$prefix/**", which is compatible with Git’s own remote-tracking ref layout (refs/remotes/$remote/**). Setting a prefix is also useful if you wish to track multiple projects that share a common repository. By default, the prefix is set to *origin/*.

> Note
>
> Before Git v2.0, the default prefix was "" (no prefix). This meant that SVN-tracking refs were put at "refs/remotes/*", which is incompatible with how Git’s own remote-tracking refs are organized. If you still want the old default, you can get it by passing `--prefix ""` on the command line (`--prefix=""` may not work if your Perl’s Getopt::Long is < v2.37).

--ignore-paths=<regex>

When passed to *init* or *clone* this regular expression will be preserved as a config key. See *fetch* for a description of *--ignore-paths*.

--include-paths=<regex>

When passed to *init* or *clone* this regular expression will be preserved as a config key. See *fetch* for a description of *--include-paths*.

--no-minimize-url

When tracking multiple directories (using --stdlayout, --branches, or --tags options), git svn will attempt to connect to the root (or highest allowed level) of the Subversion repository. This default allows better tracking of history if entire projects are moved within a repository, but may cause issues on repositories where read access restrictions are in place. Passing *--no-minimize-url* will allow git svn to accept URLs as-is without attempting to connect to a higher level directory. This option is off by default when only one URL/branch is tracked (it would do little good).

fetch

Fetch unfetched revisions from the Subversion remote we are tracking. The name of the [svn-remote "…"] section in the $GIT_DIR/config file may be specified as an optional command-line argument.

This automatically updates the rev*map if needed (see* $GIT*DIR/svn/\/.rev_map.** in the FILES section below for details).

--localtime

Store Git commit times in the local time zone instead of UTC. This makes *git log* (even without --date=local) show the same times that `svn log` would in the local time zone.

This doesn’t interfere with interoperating with the Subversion repository you cloned from, but if you wish for your local Git repository to be able to interoperate with someone else’s local Git repository, either don’t use this option or you should both use it in the same local time zone.

--parent

Fetch only from the SVN parent of the current HEAD.

--ignore-paths=<regex>

This allows one to specify a Perl regular expression that will cause skipping of all matching paths from checkout from SVN. The *--ignore-paths* option should match for every *fetch* (including automatic fetches due to *clone*, *dcommit*, *rebase*, etc) on a given repository.

```
config key: svn-remote.&lt;name&gt;.ignore-paths
```

If the ignore-paths configuration key is set, and the command-line option is also given, both regular expressions will be used.

Examples:

Skip "doc*" directory for every fetch

```
--ignore-paths="^doc"
```

Skip "branches" and "tags" of first level directories

```
--ignore-paths="^[^/]+/(?:branches|tags)"
```

--include-paths=<regex>

This allows one to specify a Perl regular expression that will cause the inclusion of only matching paths from checkout from SVN. The *--include-paths* option should match for every *fetch* (including automatic fetches due to *clone*, *dcommit*, *rebase*, etc) on a given repository. *--ignore-paths* takes precedence over *--include-paths*.

```
config key: svn-remote.&lt;name&gt;.include-paths
```

--log-window-size=<n>

Fetch <n> log entries per request when scanning Subversion history. The default is 100. For very large Subversion repositories, larger values may be needed for *clone*/*fetch* to complete in reasonable time. But overly large values may lead to higher memory usage and request timeouts.

clone

Runs *init* and *fetch*. It will automatically create a directory based on the basename of the URL passed to it; or if a second argument is passed; it will create a directory and work within that. It accepts all arguments that the *init* and *fetch* commands accept; with the exception of *--fetch-all* and *--parent*. After a repository is cloned, the *fetch* command will be able to update revisions without affecting the working tree; and the *rebase* command will be able to update the working tree with the latest changes.

--preserve-empty-dirs

Create a placeholder file in the local Git repository for each empty directory fetched from Subversion. This includes directories that become empty by removing all entries in the Subversion repository (but not the directory itself). The placeholder files are also tracked and removed when no longer necessary.

--placeholder-filename=<filename>

Set the name of placeholder files created by --preserve-empty-dirs. Default: ".gitignore"

rebase

This fetches revisions from the SVN parent of the current HEAD and rebases the current (uncommitted to SVN) work against it.

This works similarly to `svn update` or *git pull* except that it preserves linear history with *git rebase* instead of *git merge* for ease of dcommitting with *git svn*.

This accepts all options that *git svn fetch* and *git rebase* accept. However, *--fetch-all* only fetches from the current [svn-remote], and not all [svn-remote] definitions.

Like *git rebase*; this requires that the working tree be clean and have no uncommitted changes.

This automatically updates the rev*map if needed (see* $GIT*DIR/svn/\/.rev_map.** in the FILES section below for details).

-l

--local

Do not fetch remotely; only run *git rebase* against the last fetched commit from the upstream SVN.

dcommit

Commit each diff from the current branch directly to the SVN repository, and then rebase or reset (depending on whether or not there is a diff between SVN and head). This will create a revision in SVN for each commit in Git.

When an optional Git branch name (or a Git commit object name) is specified as an argument, the subcommand works on the specified branch, not on the current branch.

Use of *dcommit* is preferred to *set-tree* (below).

--no-rebase

After committing, do not rebase or reset.

--commit-url <URL>

Commit to this SVN URL (the full path). This is intended to allow existing *git svn* repositories created with one transport method (e.g. `svn://` or `http://` for anonymous read) to be reused if a user is later given access to an alternate transport method (e.g. `svn+ssh://` or `https://`) for commit.

```
config key: svn-remote.&lt;name&gt;.commiturl
config key: svn.commiturl (overwrites all svn-remote.&lt;name&gt;.commiturl options)
```

Note that the SVN URL of the commiturl config key includes the SVN branch. If you rather want to set the commit URL for an entire SVN repository use svn-remote.<name>.pushurl instead.

Using this option for any other purpose (don’t ask) is very strongly discouraged.

--mergeinfo=<mergeinfo>

Add the given merge information during the dcommit (e.g. `--mergeinfo="/branches/foo:1-10"`). All svn server versions can store this information (as a property), and svn clients starting from version 1.5 can make use of it. To specify merge information from multiple branches, use a single space character between the branches (`--mergeinfo="/branches/foo:1-10 /branches/bar:3,5-6,8"`)

```
config key: svn.pushmergeinfo
```

This option will cause git-svn to attempt to automatically populate the svn:mergeinfo property in the SVN repository when possible. Currently, this can only be done when dcommitting non-fast-forward merges where all parents but the first have already been pushed into SVN.

--interactive

Ask the user to confirm that a patch set should actually be sent to SVN. For each patch, one may answer "yes" (accept this patch), "no" (discard this patch), "all" (accept all patches), or "quit".

git svn dcommit returns immediately if answer is "no" or "quit", without committing anything to SVN.

branch

Create a branch in the SVN repository.

-m

--message

Allows to specify the commit message.

-t

--tag

Create a tag by using the tags_subdir instead of the branches_subdir specified during git svn init.

-d<path>

--destination=<path>

If more than one --branches (or --tags) option was given to the *init* or *clone* command, you must provide the location of the branch (or tag) you wish to create in the SVN repository. <path> specifies which path to use to create the branch or tag and should match the pattern on the left-hand side of one of the configured branches or tags refspecs. You can see these refspecs with the commands

```
git config --get-all svn-remote.&lt;name&gt;.branches
git config --get-all svn-remote.&lt;name&gt;.tags
```

where <name> is the name of the SVN repository as specified by the -R option to *init* (or "svn" by default).

--username

Specify the SVN username to perform the commit as. This option overrides the *username* configuration property.

--commit-url

Use the specified URL to connect to the destination Subversion repository. This is useful in cases where the source SVN repository is read-only. This option overrides configuration property *commiturl*.

```
git config --get-all svn-remote.&lt;name&gt;.commiturl
```

--parents

Create parent folders. This parameter is equivalent to the parameter --parents on svn cp commands and is useful for non-standard repository layouts.

tag

Create a tag in the SVN repository. This is a shorthand for *branch -t*.

log

This should make it easy to look up svn log messages when svn users refer to -r/--revision numbers.

The following features from ‘svn log’ are supported:

-r <n>[:<n>]

--revision=<n>[:<n>]

is supported, non-numeric args are not: HEAD, NEXT, BASE, PREV, etc …

-v

--verbose

it’s not completely compatible with the --verbose output in svn log, but reasonably close.

--limit=<n>

is NOT the same as --max-count, doesn’t count merged/excluded commits

--incremental

supported

New features:

--show-commit

shows the Git commit sha1, as well

--oneline

our version of --pretty=oneline

> Note
>
> SVN itself only stores times in UTC and nothing else. The regular svn client converts the UTC time to the local time (or based on the TZ= environment). This command has the same behaviour.

Any other arguments are passed directly to *git log*

blame

Show what revision and author last modified each line of a file. The output of this mode is format-compatible with the output of ‘svn blame’ by default. Like the SVN blame command, local uncommitted changes in the working tree are ignored; the version of the file in the HEAD revision is annotated. Unknown arguments are passed directly to *git blame*.

--git-format

Produce output in the same format as *git blame*, but with SVN revision numbers instead of Git commit hashes. In this mode, changes that haven’t been committed to SVN (including local working-copy edits) are shown as revision 0.

find-rev

When given an SVN revision number of the form *rN*, returns the corresponding Git commit hash (this can optionally be followed by a tree-ish to specify which branch should be searched). When given a tree-ish, returns the corresponding SVN revision number.

-B

--before

Don’t require an exact match if given an SVN revision, instead find the commit corresponding to the state of the SVN repository (on the current branch) at the specified revision.

-A

--after

Don’t require an exact match if given an SVN revision; if there is not an exact match return the closest match searching forward in the history.

set-tree

You should consider using *dcommit* instead of this command. Commit specified commit or tree objects to SVN. This relies on your imported fetch data being up-to-date. This makes absolutely no attempts to do patching when committing to SVN, it simply overwrites files with those specified in the tree or commit. All merging is assumed to have taken place independently of *git svn* functions.

create-ignore

Recursively finds the svn:ignore property on directories and creates matching .gitignore files. The resulting files are staged to be committed, but are not committed. Use -r/--revision to refer to a specific revision.

show-ignore

Recursively finds and lists the svn:ignore property on directories. The output is suitable for appending to the $GIT_DIR/info/exclude file.

mkdirs

Attempts to recreate empty directories that core Git cannot track based on information in $GIT_DIR/svn/<refname>/unhandled.log files. Empty directories are automatically recreated when using "git svn clone" and "git svn rebase", so "mkdirs" is intended for use after commands like "git checkout" or "git reset". (See the svn-remote.<name>.automkdirs config file option for more information.)

commit-diff

Commits the diff of two tree-ish arguments from the command-line. This command does not rely on being inside an `git svn init`-ed repository. This command takes three arguments, (a) the original tree to diff against, (b) the new tree result, (c) the URL of the target Subversion repository. The final argument (URL) may be omitted if you are working from a *git svn*-aware repository (that has been `init`-ed with *git svn*). The -r<revision> option is required for this.

info

Shows information about a file or directory similar to what ‘svn info’ provides. Does not currently support a -r/--revision argument. Use the --url option to output only the value of the *URL:* field.

proplist

Lists the properties stored in the Subversion repository about a given file or directory. Use -r/--revision to refer to a specific Subversion revision.

propget

Gets the Subversion property given as the first argument, for a file. A specific revision can be specified with -r/--revision.

show-externals

Shows the Subversion externals. Use -r/--revision to specify a specific revision.

gc

Compress $GIT_DIR/svn/<refname>/unhandled.log files and remove $GIT_DIR/svn/<refname>/index files.

reset

Undoes the effects of *fetch* back to the specified revision. This allows you to re-*fetch* an SVN revision. Normally the contents of an SVN revision should never change and *reset* should not be necessary. However, if SVN permissions change, or if you alter your --ignore-paths option, a *fetch* may fail with "not found in commit" (file not previously visible) or "checksum mismatch" (missed a modification). If the problem file cannot be ignored forever (with --ignore-paths) the only way to repair the repo is to use *reset*.

Only the rev*map and refs/remotes/git-svn are changed (see* $GIT*DIR/svn/\/.rev_map.** in the FILES section below for details). Follow *reset* with a *fetch* and then *git reset* or *git rebase* to move local branches onto the new tree.

-r <n>

--revision=<n>

Specify the most recent revision to keep. All later revisions are discarded.

-p

--parent

Discard the specified revision as well, keeping the nearest parent instead.

Example:

Assume you have local changes in "master", but you need to refetch "r2".

```
    r1---r2---r3 remotes/git-svn
                \
                 A---B master
```

Fix the ignore-paths or SVN permissions problem that caused "r2" to be incomplete in the first place. Then:

```
git svn reset -r2 -p
git svn fetch
```

```
    r1---r2'--r3' remotes/git-svn
      \
       r2---r3---A---B master
```

Then fixup "master" with *git rebase*. Do NOT use *git merge* or your history will not be compatible with a future *dcommit*!

```
git rebase --onto remotes/git-svn A^ master
```

```
    r1---r2'--r3' remotes/git-svn
                \
                 A'--B' master
```

OPTIONS

--shared[=(false|true|umask|group|all|world|everybody)]

--template=<template_directory>

Only used with the *init* command. These are passed directly to *git init*.

-r <arg>

--revision <arg>

Used with the *fetch* command.

This allows revision ranges for partial/cauterized history to be supported. $NUMBER, $NUMBER1:$NUMBER2 (numeric ranges), $NUMBER:HEAD, and BASE:$NUMBER are all supported.

This can allow you to make partial mirrors when running fetch; but is generally not recommended because history will be skipped and lost.

-

--stdin

Only used with the *set-tree* command.

Read a list of commits from stdin and commit them in reverse order. Only the leading sha1 is read from each line, so *git rev-list --pretty=oneline* output can be used.

--rmdir

Only used with the *dcommit*, *set-tree* and *commit-diff* commands.

Remove directories from the SVN tree if there are no files left behind. SVN can version empty directories, and they are not removed by default if there are no files left in them. Git cannot version empty directories. Enabling this flag will make the commit to SVN act like Git.

```
config key: svn.rmdir
```

-e

--edit

Only used with the *dcommit*, *set-tree* and *commit-diff* commands.

Edit the commit message before committing to SVN. This is off by default for objects that are commits, and forced on when committing tree objects.

```
config key: svn.edit
```

-l<num>

--find-copies-harder

Only used with the *dcommit*, *set-tree* and *commit-diff* commands.

They are both passed directly to *git diff-tree*; see git-diff-tree[1] for more information.

```
config key: svn.l
config key: svn.findcopiesharder
```

-A<filename>

--authors-file=<filename>

Syntax is compatible with the file used by *git cvsimport*:

```
    loginname = Joe User &lt;user@example.com&gt;
```

If this option is specified and *git svn* encounters an SVN committer name that does not exist in the authors-file, *git svn* will abort operation. The user will then have to add the appropriate entry. Re-running the previous *git svn* command after the authors-file is modified should continue operation.

```
config key: svn.authorsfile
```

--authors-prog=<filename>

If this option is specified, for each SVN committer name that does not exist in the authors file, the given file is executed with the committer name as the first argument. The program is expected to return a single line of the form "Name <email>", which will be treated as if included in the authors file.

-q

--quiet

Make *git svn* less verbose. Specify a second time to make it even less verbose.

-m

--merge

-s<strategy>

--strategy=<strategy>

-p

--preserve-merges

These are only used with the *dcommit* and *rebase* commands.

Passed directly to *git rebase* when using *dcommit* if a *git reset* cannot be used (see *dcommit*).

-n

--dry-run

This can be used with the *dcommit*, *rebase*, *branch* and *tag* commands.

For *dcommit*, print out the series of Git arguments that would show which diffs would be committed to SVN.

For *rebase*, display the local branch associated with the upstream svn repository associated with the current branch and the URL of svn repository that will be fetched from.

For *branch* and *tag*, display the urls that will be used for copying when creating the branch or tag.

--use-log-author

When retrieving svn commits into Git (as part of *fetch*, *rebase*, or *dcommit* operations), look for the first `From:` or `Signed-off-by:` line in the log message and use that as the author string.

--add-author-from

When committing to svn from Git (as part of *commit-diff*, *set-tree* or *dcommit* operations), if the existing log message doesn’t already have a `From:` or `Signed-off-by:` line, append a `From:` line based on the Git commit’s author string. If you use this, then `--use-log-author` will retrieve a valid author string for all commits.

ADVANCED OPTIONS

-i<GIT_SVN_ID>

--id <GIT_SVN_ID>

This sets GIT*SVN_ID (instead of using the environment). This allows the user to override the default refname to fetch from when tracking a single URL. The _log* and *dcommit* commands no longer require this switch as an argument.

-R<remote name>

--svn-remote <remote name>

Specify the [svn-remote "<remote name>"] section to use, this allows SVN multiple repositories to be tracked. Default: "svn"

--follow-parent

This option is only relevant if we are tracking branches (using one of the repository layout options --trunk, --tags, --branches, --stdlayout). For each tracked branch, try to find out where its revision was copied from, and set a suitable parent in the first Git commit for the branch. This is especially helpful when we’re tracking a directory that has been moved around within the repository. If this feature is disabled, the branches created by *git svn* will all be linear and not share any history, meaning that there will be no information on where branches were branched off or merged. However, following long/convoluted histories can take a long time, so disabling this feature may speed up the cloning process. This feature is enabled by default, use --no-follow-parent to disable it.

```
config key: svn.followparent
```

CONFIG FILE-ONLY OPTIONS

svn.noMetadata

svn-remote.<name>.noMetadata

This gets rid of the *git-svn-id:* lines at the end of every commit.

This option can only be used for one-shot imports as *git svn* will not be able to fetch again without metadata. Additionally, if you lose your *$GIT_DIR/svn/\/.rev_map.** files, *git svn* will not be able to rebuild them.

The *git svn log* command will not work on repositories using this, either. Using this conflicts with the *useSvmProps* option for (hopefully) obvious reasons.

This option is NOT recommended as it makes it difficult to track down old references to SVN revision numbers in existing documentation, bug reports and archives. If you plan to eventually migrate from SVN to Git and are certain about dropping SVN history, consider git-filter-branch[1] instead. filter-branch also allows reformatting of metadata for ease-of-reading and rewriting authorship info for non-"svn.authorsFile" users.

svn.useSvmProps

svn-remote.<name>.useSvmProps

This allows *git svn* to re-map repository URLs and UUIDs from mirrors created using SVN::Mirror (or svk) for metadata.

If an SVN revision has a property, "svm:headrev", it is likely that the revision was created by SVN::Mirror (also used by SVK). The property contains a repository UUID and a revision. We want to make it look like we are mirroring the original URL, so introduce a helper function that returns the original identity URL and UUID, and use it when generating metadata in commit messages.

svn.useSvnsyncProps

svn-remote.<name>.useSvnsyncprops

Similar to the useSvmProps option; this is for users of the svnsync(1) command distributed with SVN 1.4.x and later.

svn-remote.<name>.rewriteRoot

This allows users to create repositories from alternate URLs. For example, an administrator could run *git svn* on the server locally (accessing via file://) but wish to distribute the repository with a public http:// or svn:// URL in the metadata so users of it will see the public URL.

svn-remote.<name>.rewriteUUID

Similar to the useSvmProps option; this is for users who need to remap the UUID manually. This may be useful in situations where the original UUID is not available via either useSvmProps or useSvnsyncProps.

svn-remote.<name>.pushurl

Similar to Git’s *remote.<name>.pushurl*, this key is designed to be used in cases where *url* points to an SVN repository via a read-only transport, to provide an alternate read/write transport. It is assumed that both keys point to the same repository. Unlike *commiturl*, *pushurl* is a base path. If either *commiturl* or *pushurl* could be used, *commiturl* takes precedence.

svn.brokenSymlinkWorkaround

This disables potentially expensive checks to workaround broken symlinks checked into SVN by broken clients. Set this option to "false" if you track a SVN repository with many empty blobs that are not symlinks. This option may be changed while *git svn* is running and take effect on the next revision fetched. If unset, *git svn* assumes this option to be "true".

svn.pathnameencoding

This instructs git svn to recode pathnames to a given encoding. It can be used by windows users and by those who work in non-utf8 locales to avoid corrupted file names with non-ASCII characters. Valid encodings are the ones supported by Perl’s Encode module.

svn-remote.<name>.automkdirs

Normally, the "git svn clone" and "git svn rebase" commands attempt to recreate empty directories that are in the Subversion repository. If this option is set to "false", then empty directories will only be created if the "git svn mkdirs" command is run explicitly. If unset, *git svn* assumes this option to be "true".

Since the noMetadata, rewriteRoot, rewriteUUID, useSvnsyncProps and useSvmProps options all affect the metadata generated and used by *git svn*; they **must** be set in the configuration file before any history is imported and these settings should never be changed once they are set.

Additionally, only one of these options can be used per svn-remote section because they affect the *git-svn-id:* metadata line, except for rewriteRoot and rewriteUUID which can be used together.

BASIC EXAMPLES

Tracking and contributing to the trunk of a Subversion-managed project (ignoring tags and branches):

```
# Clone a repo (like git clone):
    git svn clone http://svn.example.com/project/trunk
# Enter the newly cloned directory:
    cd trunk
# You should be on master branch, double-check with 'git branch'
    git branch
# Do some work and commit locally to Git:
    git commit ...
# Something is committed to SVN, rebase your local changes against the
# latest changes in SVN:
    git svn rebase
# Now commit your changes (that were committed previously using Git) to SVN,
# as well as automatically updating your working HEAD:
    git svn dcommit
# Append svn:ignore settings to the default Git exclude file:
    git svn show-ignore >> .git/info/exclude
```

Tracking and contributing to an entire Subversion-managed project (complete with a trunk, tags and branches):

```
# Clone a repo with standard SVN directory layout (like git clone):
    git svn clone http://svn.example.com/project --stdlayout --prefix svn/
# Or, if the repo uses a non-standard directory layout:
    git svn clone http://svn.example.com/project -T tr -b branch -t tag --prefix svn/
# View all branches and tags you have cloned:
    git branch -r
# Create a new branch in SVN
    git svn branch waldo
# Reset your master to trunk (or any other branch, replacing 'trunk'
# with the appropriate name):
    git reset --hard svn/trunk
# You may only dcommit to one branch/tag/trunk at a time.  The usage
# of dcommit/rebase/show-ignore should be the same as above.
```

The initial *git svn clone* can be quite time-consuming (especially for large Subversion repositories). If multiple people (or one person with multiple machines) want to use *git svn* to interact with the same Subversion repository, you can do the initial *git svn clone* to a repository on a server and have each person clone that repository with *git clone*:

```
# Do the initial import on a server
    ssh server "cd /pub && git svn clone http://svn.example.com/project [options...]"
# Clone locally - make sure the refs/remotes/ space matches the server
    mkdir project
    cd project
    git init
    git remote add origin server:/pub/project
    git config --replace-all remote.origin.fetch '+refs/remotes/*:refs/remotes/*'
    git fetch
# Prevent fetch/pull from remote Git server in the future,
# we only want to use git svn for future updates
    git config --remove-section remote.origin
# Create a local branch from one of the branches just fetched
    git checkout -b master FETCH_HEAD
# Initialize 'git svn' locally (be sure to use the same URL and
# --stdlayout/-T/-b/-t/--prefix options as were used on server)
    git svn init http://svn.example.com/project [options...]
# Pull the latest changes from Subversion
    git svn rebase
```

REBASE VS. PULL/MERGE

Prefer to use *git svn rebase* or *git rebase*, rather than *git pull* or *git merge* to synchronize unintegrated commits with a *git svn* branch. Doing so will keep the history of unintegrated commits linear with respect to the upstream SVN repository and allow the use of the preferred *git svn dcommit* subcommand to push unintegrated commits back into SVN.

Originally, *git svn* recommended that developers pulled or merged from the *git svn* branch. This was because the author favored `git svn set-tree B` to commit a single head rather than the `git svn set-tree A..B` notation to commit multiple commits. Use of *git pull* or *git merge* with `git svn set-tree A..B` will cause non-linear history to be flattened when committing into SVN and this can lead to merge commits unexpectedly reversing previous commits in SVN.

MERGE TRACKING

While *git svn* can track copy history (including branches and tags) for repositories adopting a standard layout, it cannot yet represent merge history that happened inside git back upstream to SVN users. Therefore it is advised that users keep history as linear as possible inside Git to ease compatibility with SVN (see the CAVEATS section below).

HANDLING OF SVN BRANCHES

If *git svn* is configured to fetch branches (and --follow-branches is in effect), it sometimes creates multiple Git branches for one SVN branch, where the additional branches have names of the form *branchname@nnn* (with nnn an SVN revision number). These additional branches are created if *git svn* cannot find a parent commit for the first commit in an SVN branch, to connect the branch to the history of the other branches.

Normally, the first commit in an SVN branch consists of a copy operation. *git svn* will read this commit to get the SVN revision the branch was created from. It will then try to find the Git commit that corresponds to this SVN revision, and use that as the parent of the branch. However, it is possible that there is no suitable Git commit to serve as parent. This will happen, among other reasons, if the SVN branch is a copy of a revision that was not fetched by *git svn* (e.g. because it is an old revision that was skipped with *--revision*), or if in SVN a directory was copied that is not tracked by *git svn* (such as a branch that is not tracked at all, or a subdirectory of a tracked branch). In these cases, *git svn* will still create a Git branch, but instead of using an existing Git commit as the parent of the branch, it will read the SVN history of the directory the branch was copied from and create appropriate Git commits. This is indicated by the message "Initializing parent: <branchname>".

Additionally, it will create a special branch named *<branchname>@<SVN-Revision>*, where <SVN-Revision> is the SVN revision number the branch was copied from. This branch will point to the newly created parent commit of the branch. If in SVN the branch was deleted and later recreated from a different version, there will be multiple such branches with an @.

Note that this may mean that multiple Git commits are created for a single SVN revision.

An example: in an SVN repository with a standard trunk/tags/branches layout, a directory trunk/sub is created in r.100. In r.200, trunk/sub is branched by copying it to branches/. *git svn clone -s* will then create a branch *sub*. It will also create new Git commits for r.100 through r.199 and use these as the history of branch *sub*. Thus there will be two Git commits for each revision from r.100 to r.199 (one containing trunk/, one containing trunk/sub/). Finally, it will create a branch *sub@200* pointing to the new parent commit of branch *sub* (i.e. the commit for r.200 and trunk/sub/).

CAVEATS

For the sake of simplicity and interoperating with Subversion, it is recommended that all *git svn* users clone, fetch and dcommit directly from the SVN server, and avoid all *git clone*/*pull*/*merge*/*push* operations between Git repositories and branches. The recommended method of exchanging code between Git branches and users is *git format-patch* and *git am*, or just 'dcommit’ing to the SVN repository.

Running *git merge* or *git pull* is NOT recommended on a branch you plan to *dcommit* from because Subversion users cannot see any merges you’ve made. Furthermore, if you merge or pull from a Git branch that is a mirror of an SVN branch, *dcommit* may commit to the wrong branch.

If you do merge, note the following rule: *git svn dcommit* will attempt to commit on top of the SVN commit named in

```
git log --grep=^git-svn-id: --first-parent -1
```

You *must* therefore ensure that the most recent commit of the branch you want to dcommit to is the *first* parent of the merge. Chaos will ensue otherwise, especially if the first parent is an older commit on the same SVN branch.

git clone does not clone branches under the refs/remotes/ hierarchy or any *git svn* metadata, or config. So repositories created and managed with using *git svn* should use *rsync* for cloning, if cloning is to be done at all.

Since *dcommit* uses rebase internally, any Git branches you *git push* to before *dcommit* on will require forcing an overwrite of the existing ref on the remote repository. This is generally considered bad practice, see the git-push[1] documentation for details.

Do not use the --amend option of git-commit[1] on a change you've already dcommitted. It is considered bad practice to --amend commits you've already pushed to a remote repository for other users, and dcommit with SVN is analogous to that.

When cloning an SVN repository, if none of the options for describing the repository layout is used (--trunk, --tags, --branches, --stdlayout), *git svn clone* will create a Git repository with completely linear history, where branches and tags appear as separate directories in the working copy. While this is the easiest way to get a copy of a complete repository, for projects with many branches it will lead to a working copy many times larger than just the trunk. Thus for projects using the standard directory structure (trunk/branches/tags), it is recommended to clone with option *--stdlayout*. If the project uses a non-standard structure, and/or if branches and tags are not required, it is easiest to only clone one directory (typically trunk), without giving any repository layout options. If the full history with branches and tags is required, the options *--trunk* / *--branches* / *--tags* must be used.

When using multiple --branches or --tags, *git svn* does not automatically handle name collisions (for example, if two branches from different paths have the same name, or if a branch and a tag have the same name). In these cases, use *init* to set up your Git repository then, before your first *fetch*, edit the $GIT_DIR/config file so that the branches and tags are associated with different name spaces. For example:

```
branches = stable/*:refs/remotes/svn/stable/*
branches = debug/*:refs/remotes/svn/debug/*
```

BUGS

We ignore all SVN properties except svn:executable. Any unhandled properties are logged to $GIT_DIR/svn/<refname>/unhandled.log

Renamed and copied directories are not detected by Git and hence not tracked when committing to SVN. I do not plan on adding support for this as it's quite difficult and time-consuming to get working for all the possible corner cases (Git doesn't do it, either). Committing renamed and copied files is fully supported if they're similar enough for Git to detect them.

In SVN, it is possible (though discouraged) to commit changes to a tag (because a tag is just a directory copy, thus technically the same as a branch). When cloning an SVN repository, *git svn* cannot know if such a commit to a tag will happen in the future. Thus it acts

conservatively and imports all SVN tags as branches, prefixing the tag name with *tags/*.

CONFIGURATION

git svn stores [svn-remote] configuration information in the repository $GIT*DIR/config file. It is similar the core Git [remote] sections except _fetch* keys do not accept glob arguments; but they are instead handled by the *branches* and *tags* keys. Since some SVN repositories are oddly configured with multiple projects glob expansions such those listed below are allowed:

```
[svn-remote "project-a"]
    url = http://server.org/svn
    fetch = trunk/project-a:refs/remotes/project-a/trunk
    branches = branches/*/project-a:refs/remotes/project-a/branches/*
    branches = branches/release_*:refs/remotes/project-a/branches/release_*
    branches = branches/re*se:refs/remotes/project-a/branches/*
    tags = tags/*/project-a:refs/remotes/project-a/tags/*
```

Keep in mind that the * (asterisk) wildcard of the local ref (right of the *:*) *must* be the farthest right path component; however the remote wildcard may be anywhere as long as it's an independent path component (surrounded by */* or EOL). This type of configuration is not automatically created by *init* and should be manually entered with a text-editor or using *git config*.

Also note that only one asterisk is allowed per word. For example:

```
branches = branches/re*se:refs/remotes/project-a/branches/*
```

will match branches *release*, *rese*, *re123se*, however

```
branches = branches/re*s*e:refs/remotes/project-a/branches/*
```

will produce an error.

It is also possible to fetch a subset of branches or tags by using a comma-separated list of names within braces. For example:

```
[svn-remote "huge-project"]
    url = http://server.org/svn
    fetch = trunk/src:refs/remotes/trunk
    branches = branches/{red,green}/src:refs/remotes/project-a/branches/*
    tags = tags/{1.0,2.0}/src:refs/remotes/project-a/tags/*
```

Multiple fetch, branches, and tags keys are supported:

```
[svn-remote "messy-repo"]
    url = http://server.org/svn
    fetch = trunk/project-a:refs/remotes/project-a/trunk
    fetch = branches/demos/june-project-a-demo:refs/remotes/project-a/demos/june-demo
    branches = branches/server/*:refs/remotes/project-a/branches/*
    branches = branches/demos/2011/*:refs/remotes/project-a/2011-demos/*
    tags = tags/server/*:refs/remotes/project-a/tags/*
```

Creating a branch in such a configuration requires disambiguating which location to use using the -d or --destination flag:

```
$ git svn branch -d branches/server release-2-3-0
```

Note that git-svn keeps track of the highest revision in which a branch or tag has appeared. If the subset of branches or tags is changed after fetching, then $GIT_DIR/svn/.metadata must be manually edited to remove (or reset) branches-maxRev and/or tags-maxRev as appropriate.

FILES

$GIT_DIR/svn/\/.rev_map.*

Mapping between Subversion revision numbers and Git commit names. In a repository where the noMetadata option is not set, this can be rebuilt from the git-svn-id: lines that are at the end of every commit (see the *svn.noMetadata* section above for details).

git svn fetch and *git svn rebase* automatically update the rev*map if it is missing or not up to date. _git svn reset* automatically rewinds it.

SEE ALSO

git-rebase[1]

GIT

Part of the git[1] suite

fast-import

NAME

git-fast-import - Backend for fast Git data importers

SYNOPSIS

```
frontend | git fast-import [options]
```

DESCRIPTION

This program is usually not what the end user wants to run directly. Most end users want to use one of the existing frontend programs, which parses a specific type of foreign source and feeds the contents stored there to *git fast-import*.

fast-import reads a mixed command/data stream from standard input and writes one or more packfiles directly into the current repository. When EOF is received on standard input, fast import writes out updated branch and tag refs, fully updating the current repository with the newly imported data.

The fast-import backend itself can import into an empty repository (one that has already been initialized by *git init*) or incrementally update an existing populated repository. Whether or not incremental imports are supported from a particular foreign source depends on the frontend program in use.

OPTIONS

--force

Force updating modified existing branches, even if doing so would cause commits to be lost (as the new commit does not contain the old commit).

--quiet

Disable all non-fatal output, making fast-import silent when it is successful. This option disables the output shown by --stats.

--stats

Display some basic statistics about the objects fast-import has created, the packfiles they were stored into, and the memory used by fast-import during this run. Showing this output is currently the default, but can be disabled with --quiet.

Options for Frontends

--cat-blob-fd=<fd>

Write responses to `get-mark`, `cat-blob`, and `ls` queries to the file descriptor <fd> instead of `stdout`. Allows `progress` output intended for the end-user to be separated from other output.

--date-format=<fmt>

Specify the type of dates the frontend will supply to fast-import within `author`, `committer` and `tagger` commands. See “Date Formats” below for details about which formats are supported, and their syntax.

--done

Terminate with error if there is no `done` command at the end of the stream. This option might be useful for detecting errors that cause the frontend to terminate before it has started to write a stream.

Locations of Marks Files

--export-marks=<file>

Dumps the internal marks table to <file> when complete. Marks are written one per line as `:markid SHA-1`. Frontends can use this file to validate imports after they have been completed, or to save the marks table across incremental runs. As <file> is only opened and truncated at checkpoint (or completion) the same path can also be safely given to --import-marks.

--import-marks=<file>

Before processing any input, load the marks specified in <file>. The input file must exist, must be readable, and must use the same format as produced by --export-marks. Multiple options may be supplied to import more than one set of marks. If a mark is defined to different values, the last file wins.

--import-marks-if-exists=<file>

Like --import-marks but instead of erroring out, silently skips the file if it does not exist.

--[no-]relative-marks

After specifying --relative-marks the paths specified with --import-marks= and --export-marks= are relative to an internal directory in the current repository. In git-fast-import this means that the paths are relative to the .git/info/fast-import directory. However, other importers may use a different location.

Relative and non-relative marks may be combined by interweaving --(no-)-relative-marks with the --(import|export)-marks= options.

Performance and Compression Tuning

--active-branches=<n>

Maximum number of branches to maintain active at once. See “Memory Utilization” below for details. Default is 5.

--big-file-threshold=<n>

Maximum size of a blob that fast-import will attempt to create a delta for, expressed in bytes. The default is 512m (512 MiB). Some importers may wish to lower this on systems with constrained memory.

--depth=<n>

Maximum delta depth, for blob and tree deltification. Default is 10.

--export-pack-edges=<file>

After creating a packfile, print a line of data to <file> listing the filename of the packfile and the last commit on each branch that was written to that packfile. This information may be useful after importing projects whose total object set exceeds the 4 GiB packfile limit, as these commits can be used as edge points during calls to *git pack-objects*.

--max-pack-size=<n>

Maximum size of each output packfile. The default is unlimited.

Performance

The design of fast-import allows it to import large projects in a minimum amount of memory usage and processing time. Assuming the frontend is able to keep up with fast-import and feed it a constant stream of data, import times for projects holding 10+ years of history and containing 100,000+ individual commits are generally completed in just 1-2 hours on quite modest (~$2,000 USD) hardware.

Most bottlenecks appear to be in foreign source data access (the source just cannot extract revisions fast enough) or disk IO (fast-import writes as fast as the disk will take the data). Imports will run faster if the source data is stored on a different drive than the destination Git repository (due to less IO contention).

Development Cost

A typical frontend for fast-import tends to weigh in at approximately 200 lines of Perl/Python/Ruby code. Most developers have been able to create working importers in just a couple of hours, even though it is their first exposure to fast-import, and sometimes even to Git. This is an ideal situation, given that most conversion tools are throw-away (use once, and never look back).

Parallel Operation

Like *git push* or *git fetch*, imports handled by fast-import are safe to run alongside parallel `git repack -a -d` or `git gc` invocations, or any other Git operation (including *git prune*, as loose objects are never used by fast-import).

fast-import does not lock the branch or tag refs it is actively importing. After the import, during its ref update phase, fast-import tests each existing branch ref to verify the update will be a fast-forward update (the commit stored in the ref is contained in the new history of the commit to be written). If the update is not a fast-forward update, fast-import will skip updating that ref and instead prints a warning message. fast-import will always attempt to update all branch refs, and does not stop on the first failure.

Branch updates can be forced with --force, but it’s recommended that this only be used on an otherwise quiet repository. Using --force is not necessary for an initial import into an empty repository.

Technical Discussion

fast-import tracks a set of branches in memory. Any branch can be created or modified at any point during the import process by sending a `commit` command on the input stream. This design allows a frontend program to process an unlimited number of branches simultaneously, generating commits in the order they are available from the source data. It also simplifies the frontend programs considerably.

fast-import does not use or alter the current working directory, or any file within it. (It does however update the current Git repository, as referenced by `GIT_DIR`.) Therefore an import frontend may use the working directory for its own purposes, such as extracting file revisions from the foreign source. This ignorance of the working directory also allows fast-import to run very quickly, as it does not need to perform any costly file update operations when switching between branches.

Input Format

With the exception of raw file data (which Git does not interpret) the fast-import input format is text (ASCII) based. This text based format simplifies development and debugging of frontend programs, especially when a higher level language such as Perl, Python or Ruby is being used.

fast-import is very strict about its input. Where we say SP below we mean **exactly** one space. Likewise LF means one (and only one) linefeed and HT one (and only one) horizontal tab. Supplying additional whitespace characters will cause unexpected results, such as branch names or file names with leading or trailing spaces in their name, or early termination of fast-import when it encounters unexpected input.

Stream Comments

To aid in debugging frontends fast-import ignores any line that begins with `#` (ASCII pound/hash) up to and including the line ending `LF`. A comment line may contain any sequence of bytes that does not contain an LF and therefore may be used to include any detailed debugging information that might be specific to the frontend and useful when inspecting a fast-import data stream.

Date Formats

The following date formats are supported. A frontend should select the format it will use for this import by passing the format name in the --date-format=<fmt> command-line option.

`raw`

This is the Git native format and is `<time> SP <offutc>`. It is also fast-import's default format, if --date-format was not specified.

The time of the event is specified by `<time>` as the number of seconds since the UNIX epoch (midnight, Jan 1, 1970, UTC) and is written as an ASCII decimal integer.

The local offset is specified by `<offutc>` as a positive or negative offset from UTC. For example EST (which is 5 hours behind UTC) would be expressed in `<tz>` by “-0500” while UTC is “+0000”. The local offset does not affect `<time>`; it is used only as an advisement to help formatting routines display the timestamp.

If the local offset is not available in the source material, use “+0000”, or the most common local offset. For example many organizations have a CVS repository which has only ever been accessed by users who are located in the same location and time zone. In this case a reasonable offset from UTC could be assumed.

Unlike the `rfc2822` format, this format is very strict. Any variation in formatting will cause fast-import to reject the value.

`rfc2822`

This is the standard email format as described by RFC 2822.

An example value is “Tue Feb 6 11:22:18 2007 -0500”. The Git parser is accurate, but a little on the lenient side. It is the same parser used by *git am* when applying patches received from email.

Some malformed strings may be accepted as valid dates. In some of these cases Git will still be able to obtain the correct date from the malformed string. There are also some types of malformed strings which Git will parse wrong, and yet consider valid. Seriously malformed strings will be rejected.

Unlike the `raw` format above, the time zone/UTC offset information contained in an RFC 2822 date string is used to adjust the date value to UTC prior to storage. Therefore it is important that this information be as accurate as possible.

If the source material uses RFC 2822 style dates, the frontend should let fast-import handle the parsing and conversion (rather than attempting to do it itself) as the Git parser has been well tested in the wild.

Frontends should prefer the `raw` format if the source material already uses UNIX-epoch format, can be coaxed to give dates in that format, or its format is easily convertible to it, as there is no ambiguity in parsing.

`now`

Always use the current time and time zone. The literal `now` must always be supplied for `<when>`.

This is a toy format. The current time and time zone of this system is always copied into the identity string at the time it is being created by fast-import. There is no way to specify a different time or time zone.

This particular format is supplied as it's short to implement and may be useful to a process that wants to create a new commit right now, without needing to use a working directory or *git update-index*.

If separate `author` and `committer` commands are used in a `commit` the timestamps may not match, as the system clock will be polled twice (once for each command). The only way to ensure that both author and committer identity information has the same timestamp is to omit `author` (thus copying from `committer`) or to use a date format other than `now`.

Commands

fast-import accepts several commands to update the current repository and control the current import process. More detailed discussion (with examples) of each command follows later.

`commit`

Creates a new branch or updates an existing branch by creating a new commit and updating the branch to point at the newly created commit.

`tag`

Creates an annotated tag object from an existing commit or branch. Lightweight tags are not supported by this command, as they are not recommended for recording meaningful points in time.

`reset`

Reset an existing branch (or a new branch) to a specific revision. This command must be used to change a branch to a specific revision without making a commit on it.

`blob`

Convert raw file data into a blob, for future use in a `commit` command. This command is optional and is not needed to perform an import.

`checkpoint`

Forces fast-import to close the current packfile, generate its unique SHA-1 checksum and index, and start a new packfile. This command is optional and is not needed to perform an import.

`progress`

Causes fast-import to echo the entire line to its own standard output. This command is optional and is not needed to perform an import.

`done`

Marks the end of the stream. This command is optional unless the `done` feature was requested using the `--done` command-line option or `feature done` command.

`get-mark`

Causes fast-import to print the SHA-1 corresponding to a mark to the file descriptor set with `--cat-blob-fd`, or `stdout` if unspecified.

`cat-blob`

Causes fast-import to print a blob in *cat-file --batch* format to the file descriptor set with `--cat-blob-fd` or `stdout` if unspecified.

`ls`

Causes fast-import to print a line describing a directory entry in *ls-tree* format to the file descriptor set with `--cat-blob-fd` or `stdout` if unspecified.

`feature`

Enable the specified feature. This requires that fast-import supports the specified feature, and aborts if it does not.

`option`

Specify any of the options listed under OPTIONS that do not change stream semantic to suit the frontend’s needs. This command is optional and is not needed to perform an import.

`commit`

Create or update a branch with a new commit, recording one logical change to the project.

```
    'commit' SP <ref> LF
    mark?
    ('author' (SP <name>)? SP LT <email> GT SP <when> LF)?
    'committer' (SP <name>)? SP LT <email> GT SP <when> LF
    data
    ('from' SP <commit-ish> LF)?
    ('merge' SP <commit-ish> LF)?
    (filemodify | filedelete | filecopy | filerename | filedeleteall | notemodify)*
    LF?
```

where `<ref>` is the name of the branch to make the commit on. Typically branch names are prefixed with `refs/heads/` in Git, so importing the CVS branch symbol `RELENG-1_0` would use `refs/heads/RELENG-1_0` for the value of `<ref>`. The value of `<ref>` must be a valid refname in Git. As `LF` is not valid in a Git refname, no quoting or escaping syntax is supported here.

A `mark` command may optionally appear, requesting fast-import to save a reference to the newly created commit for future use by the frontend (see below for format). It is very common for frontends to mark every commit they create, thereby allowing future branch

creation from any imported commit.

The `data` command following `committer` must supply the commit message (see below for `data` command syntax). To import an empty commit message use a 0 length data. Commit messages are free-form and are not interpreted by Git. Currently they must be encoded in UTF-8, as fast-import does not permit other encodings to be specified.

Zero or more `filemodify` , `filedelete` , `filecopy` , `filerename` , `filedeleteall` and `notemodify` commands may be included to update the contents of the branch prior to creating the commit. These commands may be supplied in any order. However it is recommended that a `filedeleteall` command precede all `filemodify` , `filecopy` , `filerename` and `notemodify` commands in the same commit, as `filedeleteall` wipes the branch clean (see below).

The `LF` after the command is optional (it used to be required).

`author`

An `author` command may optionally appear, if the author information might differ from the committer information. If `author` is omitted then fast-import will automatically use the committer's information for the author portion of the commit. See below for a description of the fields in `author` , as they are identical to `committer` .

`committer`

The `committer` command indicates who made this commit, and when they made it.

Here `<name>` is the person's display name (for example “Com M Itter”) and `<email>` is the person's email address (“cm@example.com”). `LT` and `GT` are the literal less-than (\x3c) and greater-than (\x3e) symbols. These are required to delimit the email address from the other fields in the line. Note that `<name>` and `<email>` are free-form and may contain any sequence of bytes, except `LT` , `GT` and `LF` . `<name>` is typically UTF-8 encoded.

The time of the change is specified by `<when>` using the date format that was selected by the --date-format=<fmt> command-line option. See “Date Formats” above for the set of supported formats, and their syntax.

`from`

The `from` command is used to specify the commit to initialize this branch from. This revision will be the first ancestor of the new commit. The state of the tree built at this commit will begin with the state at the `from` commit, and be altered by the content modifications in this commit.

Omitting the `from` command in the first commit of a new branch will cause fast-import to create that commit with no ancestor. This tends to be desired only for the initial commit of a project. If the frontend creates all files from scratch when making a new branch, a `merge` command may be used instead of `from` to start the commit with an empty tree. Omitting the `from` command on existing branches is usually desired, as the current commit on that branch is automatically assumed to be the first ancestor of the new commit.

As `LF` is not valid in a Git refname or SHA-1 expression, no quoting or escaping syntax is supported within `<commit-ish>`.

Here `<commit-ish>` is any of the following:

- The name of an existing branch already in fast-import's internal branch table. If fast-import doesn't know the name, it's treated as a SHA-1 expression.
- A mark reference, `:<idnum>`, where `<idnum>` is the mark number.

  The reason fast-import uses `:` to denote a mark reference is this character is not legal in a Git branch name. The leading `:` makes it easy to distinguish between the mark 42 (`:42`) and the branch 42 (`42` or `refs/heads/42`), or an abbreviated SHA-1 which happened to consist only of base-10 digits.

  Marks must be declared (via `mark`) before they can be used.
- A complete 40 byte or abbreviated commit SHA-1 in hex.
- Any valid Git SHA-1 expression that resolves to a commit. See "SPECIFYING REVISIONS" in gitrevisions[7] for details.
- The special null SHA-1 (40 zeros) specifies that the branch is to be removed.

The special case of restarting an incremental import from the current branch value should be written as:

```
from refs/heads/branch^0
```

The `^0` suffix is necessary as fast-import does not permit a branch to start from itself, and the branch is created in memory before the `from` command is even read from the input. Adding `^0` will force fast-import to resolve the commit through Git's revision parsing library, rather than its internal branch table, thereby loading in the existing value of the branch.

`merge`

Includes one additional ancestor commit. The additional ancestry link does not change the way the tree state is built at this commit. If the `from` command is omitted when creating a new branch, the first `merge` commit will be the first ancestor of the current commit, and the

branch will start out with no files. An unlimited number of `merge` commands per commit are permitted by fast-import, thereby establishing an n-way merge.

Here `<commit-ish>` is any of the commit specification expressions also accepted by `from` (see above).

`filemodify`

Included in a `commit` command to add a new file or change the content of an existing file. This command has two different means of specifying the content of the file.

External data format

The data content for the file was already supplied by a prior `blob` command. The frontend just needs to connect it.

```
'M' SP &lt;mode&gt; SP &lt;dataref&gt; SP &lt;path&gt; LF
```

Here usually `<dataref>` must be either a mark reference (`:<idnum>`) set by a prior `blob` command, or a full 40-byte SHA-1 of an existing Git blob object. If `<mode>` is
`040000`` then `<dataref>` must be the full 40-byte SHA-1 of an existing Git tree object or a`
--import-marks`.

Inline data format

The data content for the file has not been supplied yet. The frontend wants to supply it as part of this modify command.

```
'M' SP &lt;mode&gt; SP 'inline' SP &lt;path&gt; LF
data
```

See below for a detailed description of the `data` command.

In both formats `<mode>` is the type of file entry, specified in octal. Git only supports the following modes:

- `100644` or `644` : A normal (not-executable) file. The majority of files in most projects use this mode. If in doubt, this is what you want.
- `100755` or `755` : A normal, but executable, file.
- `120000` : A symlink, the content of the file will be the link target.

- `160000` : A gitlink, SHA-1 of the object refers to a commit in another repository. Git links can only be specified by SHA or through a commit mark. They are used to implement submodules.
- `040000` : A subdirectory. Subdirectories can only be specified by SHA or through a tree mark set with `--import-marks` .

In both formats `<path>` is the complete path of the file to be added (if not already existing) or modified (if already existing).

A `<path>` string must use UNIX-style directory separators (forward slash `/`), may contain any byte other than `LF` , and must not start with double quote (`"`).

A path can use C-style string quoting; this is accepted in all cases and mandatory if the filename starts with double quote or contains `LF` . In C-style quoting, the complete name should be surrounded with double quotes, and any `LF` , backslash, or double quote characters must be escaped by preceding them with a backslash (e.g., `"path/with\n, \\ and \" in it"`).

The value of `<path>` must be in canonical form. That is it must not:

- contain an empty directory component (e.g. `foo//bar` is invalid),
- end with a directory separator (e.g. `foo/` is invalid),
- start with a directory separator (e.g. `/foo` is invalid),
- contain the special component `.` or `..` (e.g. `foo/./bar` and `foo/../bar` are invalid).

The root of the tree can be represented by an empty string as `<path>` .

It is recommended that `<path>` always be encoded using UTF-8.

`filedelete`

Included in a `commit` command to remove a file or recursively delete an entire directory from the branch. If the file or directory removal makes its parent directory empty, the parent directory will be automatically removed too. This cascades up the tree until the first non-empty directory or the root is reached.

```
	'D' SP <path> LF
```

here `<path>` is the complete path of the file or subdirectory to be removed from the branch. See `filemodify` above for a detailed description of `<path>` .

`filecopy`

Recursively copies an existing file or subdirectory to a different location within the branch. The existing file or directory must exist. If the destination exists it will be completely replaced by the content copied from the source.

```
'C' SP <path> SP <path> LF
```

here the first `<path>` is the source location and the second `<path>` is the destination. See `filemodify` above for a detailed description of what `<path>` may look like. To use a source path that contains SP the path must be quoted.

A `filecopy` command takes effect immediately. Once the source location has been copied to the destination any future commands applied to the source location will not impact the destination of the copy.

`filerename`

Renames an existing file or subdirectory to a different location within the branch. The existing file or directory must exist. If the destination exists it will be replaced by the source directory.

```
'R' SP <path> SP <path> LF
```

here the first `<path>` is the source location and the second `<path>` is the destination. See `filemodify` above for a detailed description of what `<path>` may look like. To use a source path that contains SP the path must be quoted.

A `filerename` command takes effect immediately. Once the source location has been renamed to the destination any future commands applied to the source location will create new files there and not impact the destination of the rename.

Note that a `filerename` is the same as a `filecopy` followed by a `filedelete` of the source location. There is a slight performance advantage to using `filerename`, but the advantage is so small that it is never worth trying to convert a delete/add pair in source material into a rename for fast-import. This `filerename` command is provided just to simplify frontends that already have rename information and don’t want bother with decomposing it into a `filecopy` followed by a `filedelete`.

`filedeleteall`

Included in a `commit` command to remove all files (and also all directories) from the branch. This command resets the internal branch structure to have no files in it, allowing the frontend to subsequently add all interesting files from scratch.

```
'deleteall' LF
```

This command is extremely useful if the frontend does not know (or does not care to know) what files are currently on the branch, and therefore cannot generate the proper `filedelete` commands to update the content.

Issuing a `filedeleteall` followed by the needed `filemodify` commands to set the correct content will produce the same results as sending only the needed `filemodify` and `filedelete` commands. The `filedeleteall` approach may however require fast-import to use slightly more memory per active branch (less than 1 MiB for even most large projects); so frontends that can easily obtain only the affected paths for a commit are encouraged to do so.

`notemodify`

Included in a `commit` `<notes_ref>` command to add a new note annotating a `<commit-ish>` or change this annotation contents. Internally it is similar to filemodify 100644 on `<commit-ish>` path (maybe split into subdirectories). It's not advised to use any other commands to write to the `<notes_ref>` tree except `filedeleteall` to delete all existing notes in this tree. This command has two different means of specifying the content of the note.

External data format

The data content for the note was already supplied by a prior `blob` command. The frontend just needs to connect it to the commit that is to be annotated.

```
'N' SP &lt;dataref&gt; SP &lt;commit-ish&gt; LF
```

Here `<dataref>` can be either a mark reference (`:<idnum>`) set by a prior `blob` command, or a full 40-byte SHA-1 of an existing Git blob object.

Inline data format

The data content for the note has not been supplied yet. The frontend wants to supply it as part of this modify command.

```
'N' SP 'inline' SP &lt;commit-ish&gt; LF
data
```

See below for a detailed description of the `data` command.

In both formats `<commit-ish>` is any of the commit specification expressions also accepted by `from` (see above).

`mark`

Arranges for fast-import to save a reference to the current object, allowing the frontend to recall this object at a future point in time, without knowing its SHA-1. Here the current object is the object creation command the `mark` command appears within. This can be `commit`, `tag`, and `blob`, but `commit` is the most common usage.

```
'mark' SP ':' <idnum> LF
```

where `<idnum>` is the number assigned by the frontend to this mark. The value of `<idnum>` is expressed as an ASCII decimal integer. The value 0 is reserved and cannot be used as a mark. Only values greater than or equal to 1 may be used as marks.

New marks are created automatically. Existing marks can be moved to another object simply by reusing the same `<idnum>` in another `mark` command.

`tag`

Creates an annotated tag referring to a specific commit. To create lightweight (non-annotated) tags see the `reset` command below.

```
'tag' SP <name> LF
'from' SP <commit-ish> LF
'tagger' (SP <name>)? SP LT <email> GT SP <when> LF
data
```

where `<name>` is the name of the tag to create.

Tag names are automatically prefixed with `refs/tags/` when stored in Git, so importing the CVS branch symbol `RELENG-1_0-FINAL` would use just `RELENG-1_0-FINAL` for `<name>`, and fast-import will write the corresponding ref as `refs/tags/RELENG-1_0-FINAL`.

The value of `<name>` must be a valid refname in Git and therefore may contain forward slashes. As `LF` is not valid in a Git refname, no quoting or escaping syntax is supported here.

The `from` command is the same as in the `commit` command; see above for details.

The `tagger` command uses the same format as `committer` within `commit`; again see above for details.

The `data` command following `tagger` must supply the annotated tag message (see below for `data` command syntax). To import an empty tag message use a 0 length data. Tag messages are free-form and are not interpreted by Git. Currently they must be encoded in UTF-8, as fast-import does not permit other encodings to be specified.

Signing annotated tags during import from within fast-import is not supported. Trying to include your own PGP/GPG signature is not recommended, as the frontend does not (easily) have access to the complete set of bytes which normally goes into such a signature. If signing is required, create lightweight tags from within fast-import with `reset`, then create the annotated versions of those tags offline with the standard *git tag* process.

`reset`

Creates (or recreates) the named branch, optionally starting from a specific revision. The reset command allows a frontend to issue a new `from` command for an existing branch, or to create a new branch from an existing commit without creating a new commit.

```
        'reset' SP <ref> LF
        ('from' SP <commit-ish> LF)?
        LF?
```

For a detailed description of `<ref>` and `<commit-ish>` see above under `commit` and `from`.

The `LF` after the command is optional (it used to be required).

The `reset` command can also be used to create lightweight (non-annotated) tags. For example:

```
reset refs/tags/938
from :938
```

would create the lightweight tag `refs/tags/938` referring to whatever commit mark `:938` references.

`blob`

Requests writing one file revision to the packfile. The revision is not connected to any commit; this connection must be formed in a subsequent `commit` command by referencing the blob through an assigned mark.

```
        'blob' LF
        mark?
        data
```

The mark command is optional here as some frontends have chosen to generate the Git SHA-1 for the blob on their own, and feed that directly to `commit`. This is typically more work than it’s worth however, as marks are inexpensive to store and easy to use.

`data`

Supplies raw data (for use as blob/file content, commit messages, or annotated tag messages) to fast-import. Data can be supplied using an exact byte count or delimited with a terminating line. Real frontends intended for production-quality conversions should always use the exact byte count format, as it is more robust and performs better. The delimited format is intended primarily for testing fast-import.

Comment lines appearing within the `<raw>` part of `data` commands are always taken to be part of the body of the data and are therefore never ignored by fast-import. This makes it safe to import any file/message content whose lines might start with `#`.

Exact byte count format

The frontend must specify the number of bytes of data.

```
'data' SP &lt;count&gt; LF
&lt;raw&gt; LF?
```

where `<count>` is the exact number of bytes appearing within `<raw>`. The value of `<count>` is expressed as an ASCII decimal integer. The `LF` on either side of `<raw>` is not included in `<count>` and will not be included in the imported data.

The `LF` after `<raw>` is optional (it used to be required) but recommended. Always including it makes debugging a fast-import stream easier as the next command always starts in column 0 of the next line, even if `<raw>` did not end with an `LF`.

Delimited format

A delimiter string is used to mark the end of the data. fast-import will compute the length by searching for the delimiter. This format is primarily useful for testing and is not recommended for real data.

```
'data' SP '&lt;&lt;' &lt;delim&gt; LF
&lt;raw&gt; LF
&lt;delim&gt; LF
LF?
```

where `<delim>` is the chosen delimiter string. The string `<delim>` must not appear on a line by itself within `<raw>`, as otherwise fast-import will think the data ends earlier than it really does. The `LF` immediately trailing `<raw>` is part of `<raw>`. This is one of the limitations of the delimited format, it is impossible to supply a data chunk which does not have an LF as its last byte.

The `LF` after `<delim> LF` is optional (it used to be required).

`checkpoint`

Forces fast-import to close the current packfile, start a new one, and to save out all current branch refs, tags and marks.

```
'checkpoint' LF
LF?
```

Note that fast-import automatically switches packfiles when the current packfile reaches --max-pack-size, or 4 GiB, whichever limit is smaller. During an automatic packfile switch fast-import does not update the branch refs, tags or marks.

As a `checkpoint` can require a significant amount of CPU time and disk IO (to compute the overall pack SHA-1 checksum, generate the corresponding index file, and update the refs) it can easily take several minutes for a single `checkpoint` command to complete.

Frontends may choose to issue checkpoints during extremely large and long running imports, or when they need to allow another Git process access to a branch. However given that a 30 GiB Subversion repository can be loaded into Git through fast-import in about 3 hours, explicit checkpointing may not be necessary.

The `LF` after the command is optional (it used to be required).

`progress`

Causes fast-import to print the entire `progress` line unmodified to its standard output channel (file descriptor 1) when the command is processed from the input stream. The command otherwise has no impact on the current import, or on any of fast-import's internal state.

```
'progress' SP <any> LF
LF?
```

The `<any>` part of the command may contain any sequence of bytes that does not contain `LF`. The `LF` after the command is optional. Callers may wish to process the output through a tool such as sed to remove the leading part of the line, for example:

```
frontend | git fast-import | sed 's/^progress //'
```

Placing a `progress` command immediately after a `checkpoint` will inform the reader when the `checkpoint` has been completed and it can safely access the refs that fast-import updated.

`get-mark`

Causes fast-import to print the SHA-1 corresponding to a mark to stdout or to the file descriptor previously arranged with the `--cat-blob-fd` argument. The command otherwise has no impact on the current import; its purpose is to retrieve SHA-1s that later commits might want to refer to in their commit messages.

```
	'get-mark' SP ':' <idnum> LF
```

This command can be used anywhere in the stream that comments are accepted. In particular, the `get-mark` command can be used in the middle of a commit but not in the middle of a `data` command.

See “Responses To Commands” below for details about how to read this output safely.

`cat-blob`

Causes fast-import to print a blob to a file descriptor previously arranged with the `--cat-blob-fd` argument. The command otherwise has no impact on the current import; its main purpose is to retrieve blobs that may be in fast-import’s memory but not accessible from the target repository.

```
	'cat-blob' SP <dataref> LF
```

The `<dataref>` can be either a mark reference (`:<idnum>`) set previously or a full 40-byte SHA-1 of a Git blob, preexisting or ready to be written.

Output uses the same format as `git cat-file --batch` :

```
<sha1> SP 'blob' SP <size> LF
<contents> LF
```

This command can be used anywhere in the stream that comments are accepted. In particular, the `cat-blob` command can be used in the middle of a commit but not in the middle of a `data` command.

See “Responses To Commands” below for details about how to read this output safely.

`ls`

Prints information about the object at a path to a file descriptor previously arranged with the `--cat-blob-fd` argument. This allows printing a blob from the active commit (with `cat-blob`) or copying a blob or tree from a previous commit for use in the current one (with

`filemodify`).

The `ls` command can be used anywhere in the stream that comments are accepted, including the middle of a commit.

Reading from the active commit

This form can only be used in the middle of a `commit` . The path names a directory entry within fast-import’s active commit. The path must be quoted in this case.

```
    'ls' SP &lt;path&gt; LF
```

Reading from a named tree

The `<dataref>` can be a mark reference (`:<idnum>`) or the full 40-byte SHA-1 of a Git tag, commit, or tree object, preexisting or waiting to be written. The path is relative to the top level of the tree named by `<dataref>` .

```
    'ls' SP &lt;dataref&gt; SP &lt;path&gt; LF
```

See `filemodify` above for a detailed description of `<path>` .

Output uses the same format as `git ls-tree <tree> -- <path>` :

```
<mode> SP ('blob' | 'tree' | 'commit') SP <dataref> HT <path> LF
```

The <dataref> represents the blob, tree, or commit object at <path> and can be used in later *get-mark*, *cat-blob*, *filemodify*, or *ls* commands.

If there is no file or subtree at that path, *git fast-import* will instead report

```
missing SP <path> LF
```

See “Responses To Commands” below for details about how to read this output safely.

`feature`

Require that fast-import supports the specified feature, or abort if it does not.

```
    'feature' SP <feature> ('=' <argument>)? LF
```

The <feature> part of the command may be any one of the following:

date-format

export-marks

relative-marks

no-relative-marks

force

Act as though the corresponding command-line option with a leading -- was passed on the command line (see OPTIONS, above).

import-marks

import-marks-if-exists

Like --import-marks except in two respects: first, only one "feature import-marks" or "feature import-marks-if-exists" command is allowed per stream; second, an --import-marks= or --import-marks-if-exists command-line option overrides any of these "feature" commands in the stream; third, "feature import-marks-if-exists" like a corresponding command-line option silently skips a nonexistent file.

get-mark

cat-blob

ls

Require that the backend support the *get-mark*, *cat-blob*, or *ls* command respectively. Versions of fast-import not supporting the specified command will exit with a message indicating so. This lets the import error out early with a clear message, rather than wasting time on the early part of an import before the unsupported command is detected.

notes

Require that the backend support the *notemodify* (N) subcommand to the *commit* command. Versions of fast-import not supporting notes will exit with a message indicating so.

done

Error out if the stream ends without a *done* command. Without this feature, errors causing the frontend to end abruptly at a convenient point in the stream can go undetected. This may occur, for example, if an import front end dies in mid-operation without emitting SIGTERM or SIGKILL at its subordinate git fast-import instance.

`option`

Processes the specified option so that git fast-import behaves in a way that suits the frontend's needs. Note that options specified by the frontend are overridden by any options the user may specify to git fast-import itself.

```
    'option' SP <option> LF
```

The `<option>` part of the command may contain any of the options listed in the OPTIONS section that do not change import semantics, without the leading -- and is treated in the same way.

Option commands must be the first commands on the input (not counting feature commands), to give an option command after any non-option command is an error.

The following command-line options change import semantics and may therefore not be passed as option:

- date-format
- import-marks
- export-marks
- cat-blob-fd
- force

`done`

If the `done` feature is not in use, treated as if EOF was read. This can be used to tell fast-import to finish early.

If the `--done` command-line option or `feature done` command is in use, the `done` command is mandatory and marks the end of the stream.

Responses To Commands

New objects written by fast-import are not available immediately. Most fast-import commands have no visible effect until the next checkpoint (or completion). The frontend can send commands to fill fast-import's input pipe without worrying about how quickly they will take effect, which improves performance by simplifying scheduling.

For some frontends, though, it is useful to be able to read back data from the current repository as it is being updated (for example when the source material describes objects in terms of patches to be applied to previously imported objects). This can be accomplished by connecting the frontend and fast-import via bidirectional pipes:

```
mkfifo fast-import-output
frontend <fast-import-output |
git fast-import >fast-import-output
```

A frontend set up this way can use `progress`, `get-mark`, `ls`, and `cat-blob` commands to read information from the import in progress.

To avoid deadlock, such frontends must completely consume any pending output from `progress`, `ls`, `get-mark`, and `cat-blob` before performing writes to fast-import that might block.

Crash Reports

If fast-import is supplied invalid input it will terminate with a non-zero exit status and create a crash report in the top level of the Git repository it was importing into. Crash reports contain a snapshot of the internal fast-import state as well as the most recent commands that lead up to the crash.

All recent commands (including stream comments, file changes and progress commands) are shown in the command history within the crash report, but raw file data and commit messages are excluded from the crash report. This exclusion saves space within the report file and reduces the amount of buffering that fast-import must perform during execution.

After writing a crash report fast-import will close the current packfile and export the marks table. This allows the frontend developer to inspect the repository state and resume the import from the point where it crashed. The modified branches and tags are not updated during a crash, as the import did not complete successfully. Branch and tag information can be found in the crash report and must be applied manually if the update is needed.

An example crash:

```
$ cat >in <<END_OF_INPUT
# my very first test commit
commit refs/heads/master
committer Shawn O. Pearce <spearce> 19283 -0400
# who is that guy anyway?
data <<EOF
this is my commit
EOF
M 644 inline .gitignore
data <<EOF
.gitignore
EOF
M 777 inline bob
END_OF_INPUT
```

```
$ git fast-import <in
fatal: Corrupt mode: M 777 inline bob
fast-import: dumping crash report to .git/fast_import_crash_8434
```

```
$ cat .git/fast_import_crash_8434
fast-import crash report:
    fast-import process: 8434
    parent process     : 1391
    at Sat Sep 1 00:58:12 2007
```

```
fatal: Corrupt mode: M 777 inline bob
```

```
Most Recent Commands Before Crash
---------------------------------
  # my very first test commit
  commit refs/heads/master
  committer Shawn O. Pearce <spearce> 19283 -0400
  # who is that guy anyway?
  data <<EOF
  M 644 inline .gitignore
  data <<EOF
* M 777 inline bob
```

```
Active Branch LRU
-----------------
    active_branches = 1 cur, 5 max
```

```
pos  clock name
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 1)      0 refs/heads/master
```

```
Inactive Branches
-----------------
refs/heads/master:
  status      : active loaded dirty
  tip commit  : 0000000000000000000000000000000000000000
  old tree    : 0000000000000000000000000000000000000000
  cur tree    : 0000000000000000000000000000000000000000
  commit clock: 0
  last pack   :
```

```
-------------------
END OF CRASH REPORT
```

Tips and Tricks

The following tips and tricks have been collected from various users of fast-import, and are offered here as suggestions.

Use One Mark Per Commit

When doing a repository conversion, use a unique mark per commit (`mark :<n>`) and supply the --export-marks option on the command line. fast-import will dump a file which lists every mark and the Git object SHA-1 that corresponds to it. If the frontend can tie the marks back to the source repository, it is easy to verify the accuracy and completeness of the import by comparing each Git commit to the corresponding source revision.

Coming from a system such as Perforce or Subversion this should be quite simple, as the fast-import mark can also be the Perforce changeset number or the Subversion revision number.

Freely Skip Around Branches

Don’t bother trying to optimize the frontend to stick to one branch at a time during an import. Although doing so might be slightly faster for fast-import, it tends to increase the complexity of the frontend code considerably.

The branch LRU builtin to fast-import tends to behave very well, and the cost of activating an inactive branch is so low that bouncing around between branches has virtually no impact on import performance.

Handling Renames

When importing a renamed file or directory, simply delete the old name(s) and modify the new name(s) during the corresponding commit. Git performs rename detection after-the-fact, rather than explicitly during a commit.

Use Tag Fixup Branches

Some other SCM systems let the user create a tag from multiple files which are not from the same commit/changeset. Or to create tags which are a subset of the files available in the repository.

Importing these tags as-is in Git is impossible without making at least one commit which “fixes up” the files to match the content of the tag. Use fast-import’s `reset` command to reset a dummy branch outside of your normal branch space to the base commit for the tag, then commit one or more file fixup commits, and finally tag the dummy branch.

For example since all normal branches are stored under `refs/heads/` name the tag fixup branch `TAG_FIXUP` . This way it is impossible for the fixup branch used by the importer to have namespace conflicts with real branches imported from the source (the name `TAG_FIXUP` is not `refs/heads/TAG_FIXUP`).

When committing fixups, consider using `merge` to connect the commit(s) which are supplying file revisions to the fixup branch. Doing so will allow tools such as *git blame* to track through the real commit history and properly annotate the source files.

After fast-import terminates the frontend will need to do `rm .git/TAG_FIXUP` to remove the dummy branch.

Import Now, Repack Later

As soon as fast-import completes the Git repository is completely valid and ready for use. Typically this takes only a very short time, even for considerably large projects (100,000+ commits).

However repacking the repository is necessary to improve data locality and access performance. It can also take hours on extremely large projects (especially if -f and a large --window parameter is used). Since repacking is safe to run alongside readers and writers, run the repack in the background and let it finish when it finishes. There is no reason to wait to explore your new Git project!

If you choose to wait for the repack, don’t try to run benchmarks or performance tests until repacking is completed. fast-import outputs suboptimal packfiles that are simply never seen in real use situations.

Repacking Historical Data

If you are repacking very old imported data (e.g. older than the last year), consider expending some extra CPU time and supplying --window=50 (or higher) when you run *git repack*. This will take longer, but will also produce a smaller packfile. You only need to expend the effort once, and everyone using your project will benefit from the smaller repository.

Include Some Progress Messages

Every once in a while have your frontend emit a `progress` message to fast-import. The contents of the messages are entirely free-form, so one suggestion would be to output the current month and year each time the current commit date moves into the next month. Your users will feel better knowing how much of the data stream has been processed.

Packfile Optimization

When packing a blob fast-import always attempts to deltify against the last blob written. Unless specifically arranged for by the frontend, this will probably not be a prior version of the same file, so the generated delta will not be the smallest possible. The resulting packfile will be compressed, but will not be optimal.

Frontends which have efficient access to all revisions of a single file (for example reading an RCS/CVS ,v file) can choose to supply all revisions of that file as a sequence of consecutive `blob` commands. This allows fast-import to deltify the different file revisions against each other, saving space in the final packfile. Marks can be used to later identify individual file revisions during a sequence of `commit` commands.

The packfile(s) created by fast-import do not encourage good disk access patterns. This is caused by fast-import writing the data in the order it is received on standard input, while Git typically organizes data within packfiles to make the most recent (current tip) data appear before historical data. Git also clusters commits together, speeding up revision traversal through better cache locality.

For this reason it is strongly recommended that users repack the repository with `git repack -a -d` after fast-import completes, allowing Git to reorganize the packfiles for faster data access. If blob deltas are suboptimal (see above) then also adding the `-f` option to force recomputation of all deltas can significantly reduce the final packfile size (30-50% smaller can be quite typical).

Memory Utilization

There are a number of factors which affect how much memory fast-import requires to perform an import. Like critical sections of core Git, fast-import uses its own memory allocators to amortize any overheads associated with malloc. In practice fast-import tends to amortize any malloc overheads to 0, due to its use of large block allocations.

per object

fast-import maintains an in-memory structure for every object written in this execution. On a 32 bit system the structure is 32 bytes, on a 64 bit system the structure is 40 bytes (due to the larger pointer sizes). Objects in the table are not deallocated until fast-import terminates. Importing 2 million objects on a 32 bit system will require approximately 64 MiB of memory.

The object table is actually a hashtable keyed on the object name (the unique SHA-1). This storage configuration allows fast-import to reuse an existing or already written object and avoid writing duplicates to the output packfile. Duplicate blobs are surprisingly common in an import, typically due to branch merges in the source.

per mark

Marks are stored in a sparse array, using 1 pointer (4 bytes or 8 bytes, depending on pointer size) per mark. Although the array is sparse, frontends are still strongly encouraged to use marks between 1 and n, where n is the total number of marks required for this import.

per branch

Branches are classified as active and inactive. The memory usage of the two classes is significantly different.

Inactive branches are stored in a structure which uses 96 or 120 bytes (32 bit or 64 bit systems, respectively), plus the length of the branch name (typically under 200 bytes), per branch. fast-import will easily handle as many as 10,000 inactive branches in under 2 MiB of memory.

Active branches have the same overhead as inactive branches, but also contain copies of every tree that has been recently modified on that branch. If subtree `include` has not been modified since the branch became active, its contents will not be loaded into memory, but if subtree `src` has been modified by a commit since the branch became active, then its contents will be loaded in memory.

As active branches store metadata about the files contained on that branch, their in-memory storage size can grow to a considerable size (see below).

fast-import automatically moves active branches to inactive status based on a simple least-recently-used algorithm. The LRU chain is updated on each `commit` command. The maximum number of active branches can be increased or decreased on the command line with --active-branches=.

per active tree

Trees (aka directories) use just 12 bytes of memory on top of the memory required for their entries (see “per active file” below). The cost of a tree is virtually 0, as its overhead amortizes out over the individual file entries.

per active file entry

Files (and pointers to subtrees) within active trees require 52 or 64 bytes (32/64 bit platforms) per entry. To conserve space, file and tree names are pooled in a common string table, allowing the filename “Makefile” to use just 16 bytes (after including the string header overhead) no matter how many times it occurs within the project.

The active branch LRU, when coupled with the filename string pool and lazy loading of subtrees, allows fast-import to efficiently import projects with 2,000+ branches and 45,114+ files in a very limited memory footprint (less than 2.7 MiB per active branch).

Signals

Sending **SIGUSR1** to the *git fast-import* process ends the current packfile early, simulating a `checkpoint` command. The impatient operator can use this facility to peek at the objects and refs from an import in progress, at the cost of some added running time and worse compression.

SEE ALSO

git-fast-export[1]

GIT

Part of the git[1] suite

Administration

clean

NAME

git-clean - Remove untracked files from the working tree

SYNOPSIS

```
git clean [-d] [-f] [-i] [-n] [-q] [-e <pattern>] [-x | -X] [--] <path>…
```

DESCRIPTION

Cleans the working tree by recursively removing files that are not under version control, starting from the current directory.

Normally, only files unknown to Git are removed, but if the *-x* option is specified, ignored files are also removed. This can, for example, be useful to remove all build products.

If any optional `<path>...` arguments are given, only those paths are affected.

OPTIONS

-d

Remove untracked directories in addition to untracked files. If an untracked directory is managed by a different Git repository, it is not removed by default. Use -f option twice if you really want to remove such a directory.

-f

--force

If the Git configuration variable clean.requireForce is not set to false, *git clean* will refuse to delete files or directories unless given -f, -n or -i. Git will refuse to delete directories with .git sub directory or file unless a second -f is given.

-i

--interactive

Show what would be done and clean files interactively. See “Interactive mode” for details.

-n

--dry-run

Don’t actually remove anything, just show what would be done.

-q

--quiet

Be quiet, only report errors, but not the files that are successfully removed.

-e <pattern>

--exclude=<pattern>

In addition to those found in .gitignore (per directory) and $GIT_DIR/info/exclude, also consider these patterns to be in the set of the ignore rules in effect.

-x

Don’t use the standard ignore rules read from .gitignore (per directory) and $GIT*DIR/info/exclude, but do still use the ignore rules given with* `-e` *options. This allows removing all untracked files, including build products. This can be used (possibly in conjunction with _git reset*) to create a pristine working directory to test a clean build.

-X

Remove only files ignored by Git. This may be useful to rebuild everything from scratch, but keep manually created files.

Interactive mode

When the command enters the interactive mode, it shows the files and directories to be cleaned, and goes into its interactive command loop.

The command loop shows the list of subcommands available, and gives a prompt "What now> ". In general, when the prompt ends with a single >, you can pick only one of the choices given and type return, like this:

```
    *** Commands ***
    1: clean                2: filter by pattern    3: select by numbers
    4: ask each             5: quit                 6: help
    What now> 1
```

You also could say `c` or `clean` above as long as the choice is unique.

The main command loop has 6 subcommands.

clean

Start cleaning files and directories, and then quit.

filter by pattern

This shows the files and directories to be deleted and issues an "Input ignore patterns>>" prompt. You can input space-separated patterns to exclude files and directories from deletion. E.g. ".*c* .h" will excludes files end with ".c" and ".h" from deletion. When you are satisfied with the filtered result, press ENTER (empty) back to the main menu.

select by numbers

This shows the files and directories to be deleted and issues an "Select items to delete>>" prompt. When the prompt ends with double >> like this, you can make more than one selection, concatenated with whitespace or comma. Also you can say ranges. E.g. "2-5 7,9" to choose 2,3,4,5,7,9 from the list. If the second number in a range is omitted, all remaining items are selected. E.g. "7-" to choose 7,8,9 from the list. You can say * to choose everything. Also when you are satisfied with the filtered result, press ENTER (empty) back to the main menu.

ask each

This will start to clean, and you must confirm one by one in order to delete items. Please note that this action is not as efficient as the above two actions.

quit

This lets you quit without do cleaning.

help

Show brief usage of interactive git-clean.

SEE ALSO

gitignore[5]

GIT

Part of the git[1] suite

gc

NAME

git-gc - Cleanup unnecessary files and optimize the local repository

SYNOPSIS

```
git gc [--aggressive] [--auto] [--quiet] [--prune=<date> | --no-prune] [--force]
```

DESCRIPTION

Runs a number of housekeeping tasks within the current repository, such as compressing file revisions (to reduce disk space and increase performance) and removing unreachable objects which may have been created from prior invocations of *git add*.

Users are encouraged to run this task on a regular basis within each repository to maintain good disk space utilization and good operating performance.

Some git commands may automatically run *git gc*; see the `--auto` flag below for details. If you know what you're doing and all you want is to disable this behavior permanently without further considerations, just do:

```
$ git config --global gc.auto 0
```

OPTIONS

--aggressive

Usually *git gc* runs very quickly while providing good disk space utilization and performance. This option will cause *git gc* to more aggressively optimize the repository at the expense of taking much more time. The effects of this optimization are persistent, so this option only needs to be used occasionally; every few hundred changesets or so.

--auto

With this option, *git gc* checks whether any housekeeping is required; if not, it exits without performing any work. Some git commands run `git gc --auto` after performing operations that could create many loose objects.

Housekeeping is required if there are too many loose objects or too many packs in the repository. If the number of loose objects exceeds the value of the `gc.auto` configuration variable, then all loose objects are combined into a single pack using `git repack -d -l`. Setting the value of `gc.auto` to 0 disables automatic packing of loose objects.

If the number of packs exceeds the value of `gc.autoPackLimit`, then existing packs (except those marked with a `.keep` file) are consolidated into a single pack by using the `-A` option of *git repack*. Setting `gc.autoPackLimit` to 0 disables automatic consolidation of packs.

--prune=<date>

Prune loose objects older than date (default is 2 weeks ago, overridable by the config variable `gc.pruneExpire`). --prune=all prunes loose objects regardless of their age (do not use --prune=all unless you know exactly what you are doing. Unless the repository is quiescent, you will lose newly created objects that haven't been anchored with the refs and end up corrupting your repository). --prune is on by default.

--no-prune

Do not prune any loose objects.

--quiet

Suppress all progress reports.

--force

Force `git gc` to run even if there may be another `git gc` instance running on this repository.

Configuration

The optional configuration variable *gc.reflogExpire* can be set to indicate how long historical entries within each branch's reflog should remain available in this repository. The setting is expressed as a length of time, for example *90 days* or *3 months*. It defaults to *90 days*.

The optional configuration variable *gc.reflogExpireUnreachable* can be set to indicate how long historical reflog entries which are not part of the current branch should remain available in this repository. These types of entries are generally created as a result of using

`git commit --amend` or `git rebase` and are the commits prior to the amend or rebase occurring. Since these changes are not part of the current project most users will want to expire them sooner. This option defaults to *30 days*.

The above two configuration variables can be given to a pattern. For example, this sets non-default expiry values only to remote-tracking branches:

```
[gc "refs/remotes/*"]
    reflogExpire = never
    reflogExpireUnreachable = 3 days
```

The optional configuration variable *gc.rerereResolved* indicates how long records of conflicted merge you resolved earlier are kept. This defaults to 60 days.

The optional configuration variable *gc.rerereUnresolved* indicates how long records of conflicted merge you have not resolved are kept. This defaults to 15 days.

The optional configuration variable *gc.packRefs* determines if *git gc* runs *git pack-refs*. This can be set to "notbare" to enable it within all non-bare repos or it can be set to a boolean value. This defaults to true.

The optional configuration variable *gc.aggressiveWindow* controls how much time is spent optimizing the delta compression of the objects in the repository when the --aggressive option is specified. The larger the value, the more time is spent optimizing the delta compression. See the documentation for the --window' option in git-repack[1] for more details. This defaults to 250.

Similarly, the optional configuration variable *gc.aggressiveDepth* controls --depth option in git-repack[1]. This defaults to 250.

The optional configuration variable *gc.pruneExpire* controls how old the unreferenced loose objects have to be before they are pruned. The default is "2 weeks ago".

Notes

git gc tries very hard to be safe about the garbage it collects. In particular, it will keep not only objects referenced by your current set of branches and tags, but also objects referenced by the index, remote-tracking branches, refs saved by *git filter-branch* in refs/original/, or reflogs (which may reference commits in branches that were later amended or rewound).

If you are expecting some objects to be collected and they aren’t, check all of those locations and decide whether it makes sense in your case to remove those references.

HOOKS

The *git gc --auto* command will run the *pre-auto-gc* hook. See githooks[5] for more information.

SEE ALSO

git-prune[1] git-reflog[1] git-repack[1] git-rerere[1]

GIT

Part of the git[1] suite

fsck

NAME

git-fsck - Verifies the connectivity and validity of the objects in the database

SYNOPSIS

```
git fsck [--tags] [--root] [--unreachable] [--cache] [--no-reflogs]
     [--[no-]full] [--strict] [--verbose] [--lost-found]
     [--[no-]dangling] [--[no-]progress] [--connectivity-only] [<object>*]
```

DESCRIPTION

Verifies the connectivity and validity of the objects in the database.

OPTIONS

<object>

An object to treat as the head of an unreachability trace.

If no objects are given, *git fsck* defaults to using the index file, all SHA-1 references in `refs` namespace, and all reflogs (unless --no-reflogs is given) as heads.

--unreachable

Print out objects that exist but that aren’t reachable from any of the reference nodes.

--[no-]dangling

Print objects that exist but that are never *directly* used (default). `--no-dangling` can be used to omit this information from the output.

--root

Report root nodes.

--tags

Report tags.

--cache

Consider any object recorded in the index also as a head node for an unreachability trace.

--no-reflogs

Do not consider commits that are referenced only by an entry in a reflog to be reachable. This option is meant only to search for commits that used to be in a ref, but now aren't, but are still in that corresponding reflog.

--full

Check not just objects in GIT_OBJECT_DIRECTORY ($GIT_DIR/objects), but also the ones found in alternate object pools listed in GIT_ALTERNATE_OBJECT_DIRECTORIES or $GIT_DIR/objects/info/alternates, and in packed Git archives found in $GIT_DIR/objects/pack and corresponding pack subdirectories in alternate object pools. This is now default; you can turn it off with --no-full.

--connectivity-only

Check only the connectivity of tags, commits and tree objects. By avoiding to unpack blobs, this speeds up the operation, at the expense of missing corrupt objects or other problematic issues.

--strict

Enable more strict checking, namely to catch a file mode recorded with g+w bit set, which was created by older versions of Git. Existing repositories, including the Linux kernel, Git itself, and sparse repository have old objects that triggers this check, but it is recommended to check new projects with this flag.

--verbose

Be chatty.

--lost-found

Write dangling objects into .git/lost-found/commit/ or .git/lost-found/other/, depending on type. If the object is a blob, the contents are written into the file, rather than its object name.

--[no-]progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless --no-progress or --verbose is specified. --progress forces progress status even if the standard error stream is not directed to a terminal.

DISCUSSION

git-fsck tests SHA-1 and general object sanity, and it does full tracking of the resulting reachability and everything else. It prints out any corruption it finds (missing or bad objects), and if you use the *--unreachable* flag it will also print out objects that exist but that aren't reachable from any of the specified head nodes (or the default set, as mentioned above).

Any corrupt objects you will have to find in backups or other archives (i.e., you can just remove them and do an *rsync* with some other site in the hopes that somebody else has the object you have corrupted).

Extracted Diagnostics

expect dangling commits - potential heads - due to lack of head information

You haven't specified any nodes as heads so it won't be possible to differentiate between un-parented commits and root nodes.

missing sha1 directory <*dir*>

The directory holding the sha1 objects is missing.

unreachable <type> <object>

The <type> object <object>, isn't actually referred to directly or indirectly in any of the trees or commits seen. This can mean that there's another root node that you're not specifying or that the tree is corrupt. If you haven't missed a root node then you might as well delete unreachable nodes since they can't be used.

missing <type> <object>

The <type> object <object>, is referred to but isn't present in the database.

dangling <type> <object>

The <type> object <object>, is present in the database but never *directly* used. A dangling commit could be a root node.

sha1 mismatch <object>

The database has an object who's sha1 doesn't match the database value. This indicates a serious data integrity problem.

Environment Variables

GIT_OBJECT_DIRECTORY

used to specify the object database root (usually $GIT_DIR/objects)

GIT_INDEX_FILE

used to specify the index file of the index

GIT_ALTERNATE_OBJECT_DIRECTORIES

used to specify additional object database roots (usually unset)

GIT

Part of the git[1] suite

reflog

NAME

git-reflog - Manage reflog information

SYNOPSIS

```
git reflog <subcommand> <options>
```

DESCRIPTION

The command takes various subcommands, and different options depending on the subcommand:

```
git reflog [show] [log-options] [<ref>]
git reflog expire [--expire=<time>] [--expire-unreachable=<time>]
    [--rewrite] [--updateref] [--stale-fix]
    [--dry-run] [--verbose] [--all | <refs>…]
git reflog delete [--rewrite] [--updateref]
    [--dry-run] [--verbose] ref@{specifier}…
git reflog exists <ref>
```

Reference logs, or "reflogs", record when the tips of branches and other references were updated in the local repository. Reflogs are useful in various Git commands, to specify the old value of a reference. For example, `HEAD@{2}` means "where HEAD used to be two moves ago", `master@{one.week.ago}` means "where master used to point to one week ago in this local repository", and so on. See gitrevisions[7] for more details.

This command manages the information recorded in the reflogs.

The "show" subcommand (which is also the default, in the absence of any subcommands) shows the log of the reference provided in the command-line (or `HEAD`, by default). The reflog covers all recent actions, and in addition the `HEAD` reflog records branch switching. `git reflog show` is an alias for `git log -g --abbrev-commit --pretty=oneline`; see git-log[1] for more information.

The "expire" subcommand prunes older reflog entries. Entries older than `expire` time, or entries older than `expire-unreachable` time and not reachable from the current tip, are removed from the reflog. This is typically not used directly by end users — instead, see git-gc[1].

The "delete" subcommand deletes single entries from the reflog. Its argument must be an *exact* entry (e.g. " `git reflog delete master@{2}` "). This subcommand is also typically not used directly by end users.

The "exists" subcommand checks whether a ref has a reflog. It exits with zero status if the reflog exists, and non-zero status if it does not.

OPTIONS

Options for `show`

`git reflog show` accepts any of the options accepted by `git log` .

Options for `expire`

--all

Process the reflogs of all references.

--expire=<time>

Prune entries older than the specified time. If this option is not specified, the expiration time is taken from the configuration setting `gc.reflogExpire` , which in turn defaults to 90 days. `--expire=all` prunes entries regardless of their age; `--expire=never` turns off pruning of reachable entries (but see `--expire-unreachable`).

--expire-unreachable=<time>

Prune entries older than `<time>` that are not reachable from the current tip of the branch. If this option is not specified, the expiration time is taken from the configuration setting `gc.reflogExpireUnreachable` , which in turn defaults to 30 days. `--expire-unreachable=all` prunes unreachable entries regardless of their age; `--expire-unreachable=never` turns off early pruning of unreachable entries (but see `--expire`).

--updateref

Update the reference to the value of the top reflog entry (i.e. <ref>@{0}) if the previous top entry was pruned. (This option is ignored for symbolic references.)

--rewrite

If a reflog entry’s predecessor is pruned, adjust its "old" SHA-1 to be equal to the "new" SHA-1 field of the entry that now precedes it.

--stale-fix

Prune any reflog entries that point to "broken commits". A broken commit is a commit that is not reachable from any of the reference tips and that refers, directly or indirectly, to a missing commit, tree, or blob object.

This computation involves traversing all the reachable objects, i.e. it has the same cost as *git prune*. It is primarily intended to fix corruption caused by garbage collecting using older versions of Git, which didn't protect objects referred to by reflogs.

-n

--dry-run

Do not actually prune any entries; just show what would have been pruned.

--verbose

Print extra information on screen.

Options for `delete`

`git reflog delete` accepts options `--updateref`, `--rewrite`, `-n`, `--dry-run`, and `--verbose`, with the same meanings as when they are used with `expire`.

GIT

Part of the git[1] suite

filter-branch

NAME

git-filter-branch - Rewrite branches

SYNOPSIS

```
git filter-branch [--env-filter <command>] [--tree-filter <command>]
	[--index-filter <command>] [--parent-filter <command>]
	[--msg-filter <command>] [--commit-filter <command>]
	[--tag-name-filter <command>] [--subdirectory-filter <directory>]
	[--prune-empty]
	[--original <namespace>] [-d <directory>] [-f | --force]
	[--] [<rev-list options>…]
```

DESCRIPTION

Lets you rewrite Git revision history by rewriting the branches mentioned in the <rev-list options>, applying custom filters on each revision. Those filters can modify each tree (e.g. removing a file or running a perl rewrite on all files) or information about each commit. Otherwise, all information (including original commit times or merge information) will be preserved.

The command will only rewrite the *positive* refs mentioned in the command line (e.g. if you pass *a..b*, only *b* will be rewritten). If you specify no filters, the commits will be recommitted without any changes, which would normally have no effect. Nevertheless, this may be useful in the future for compensating for some Git bugs or such, therefore such a usage is permitted.

NOTE: This command honors `.git/info/grafts` file and refs in the `refs/replace/` namespace. If you have any grafts or replacement refs defined, running this command will make them permanent.

WARNING! The rewritten history will have different object names for all the objects and will not converge with the original branch. You will not be able to easily push and distribute the rewritten branch on top of the original branch. Please do not use this command if you do not know the full implications, and avoid using it anyway, if a simple single commit would suffice to fix your problem. (See the "RECOVERING FROM UPSTREAM REBASE" section in git-rebase[1] for further information about rewriting published history.)

Always verify that the rewritten version is correct: The original refs, if different from the rewritten ones, will be stored in the namespace *refs/original/*.

Note that since this operation is very I/O expensive, it might be a good idea to redirect the temporary directory off-disk with the *-d* option, e.g. on tmpfs. Reportedly the speedup is very noticeable.

Filters

The filters are applied in the order as listed below. The <command> argument is always evaluated in the shell context using the *eval* command (with the notable exception of the commit filter, for technical reasons). Prior to that, the $GIT_COMMIT environment variable will be set to contain the id of the commit being rewritten. Also, GIT_AUTHOR_NAME, GIT_AUTHOR_EMAIL, GIT_AUTHOR_DATE, GIT_COMMITTER_NAME, GIT_COMMITTER_EMAIL, and GIT_COMMITTER_DATE are taken from the current commit and exported to the environment, in order to affect the author and committer identities of the replacement commit created by git-commit-tree[1] after the filters have run.

If any evaluation of <command> returns a non-zero exit status, the whole operation will be aborted.

A *map* function is available that takes an "original sha1 id" argument and outputs a "rewritten sha1 id" if the commit has been already rewritten, and "original sha1 id" otherwise; the *map* function can return several ids on separate lines if your commit filter emitted multiple commits.

OPTIONS

--env-filter <command>

This filter may be used if you only need to modify the environment in which the commit will be performed. Specifically, you might want to rewrite the author/committer name/email/time environment variables (see git-commit-tree[1] for details). Do not forget to re-export the variables.

--tree-filter <command>

This is the filter for rewriting the tree and its contents. The argument is evaluated in shell with the working directory set to the root of the checked out tree. The new tree is then used as-is (new files are auto-added, disappeared files are auto-removed - neither .gitignore files nor any other ignore rules **HAVE ANY EFFECT**!).

--index-filter <command>

This is the filter for rewriting the index. It is similar to the tree filter but does not check out the tree, which makes it much faster. Frequently used with `git rm --cached --ignore-unmatch ...`, see EXAMPLES below. For hairy cases, see git-update-index[1].

--parent-filter <command>

This is the filter for rewriting the commit's parent list. It will receive the parent string on stdin and shall output the new parent string on stdout. The parent string is in the format described in git-commit-tree[1]: empty for the initial commit, "-p parent" for a normal commit and "-p parent1 -p parent2 -p parent3 …" for a merge commit.

--msg-filter <command>

This is the filter for rewriting the commit messages. The argument is evaluated in the shell with the original commit message on standard input; its standard output is used as the new commit message.

--commit-filter <command>

This is the filter for performing the commit. If this filter is specified, it will be called instead of the *git commit-tree* command, with arguments of the form "<TREE_ID> [(-p <PARENT_COMMIT_ID>)…]" and the log message on stdin. The commit id is expected on stdout.

As a special extension, the commit filter may emit multiple commit ids; in that case, the rewritten children of the original commit will have all of them as parents.

You can use the *map* convenience function in this filter, and other convenience functions, too. For example, calling *skip_commit "$@"* will leave out the current commit (but not its changes! If you want that, use *git rebase* instead).

You can also use the `git_commit_non_empty_tree "$@"` instead of `git commit-tree "$@"` if you don't wish to keep commits with a single parent and that makes no change to the tree.

--tag-name-filter <command>

This is the filter for rewriting tag names. When passed, it will be called for every tag ref that points to a rewritten object (or to a tag object which points to a rewritten object). The original tag name is passed via standard input, and the new tag name is expected on standard output.

The original tags are not deleted, but can be overwritten; use "--tag-name-filter cat" to simply update the tags. In this case, be very careful and make sure you have the old tags backed up in case the conversion has run afoul.

Nearly proper rewriting of tag objects is supported. If the tag has a message attached, a new tag object will be created with the same message, author, and timestamp. If the tag has a signature attached, the signature will be stripped. It is by definition impossible to preserve signatures. The reason this is "nearly" proper, is because ideally if the tag did not change (points to the same object, has the same name, etc.) it should retain any signature. That is not the case, signatures will always be removed, buyer beware. There is also no support for changing the author or timestamp (or the tag message for that matter). Tags which point to other tags will be rewritten to point to the underlying commit.

--subdirectory-filter <directory>

Only look at the history which touches the given subdirectory. The result will contain that directory (and only that) as its project root. Implies Remap to ancestor.

--prune-empty

Some kind of filters will generate empty commits, that left the tree untouched. This switch allow git-filter-branch to ignore such commits. Though, this switch only applies for commits that have one and only one parent, it will hence keep merges points. Also, this option is not compatible with the use of *--commit-filter*. Though you just need to use the function *git_commit_non_empty_tree "$@"* instead of the `git commit-tree "$@"` idiom in your commit filter to make that happen.

--original <namespace>

Use this option to set the namespace where the original commits will be stored. The default value is *refs/original*.

-d <directory>

Use this option to set the path to the temporary directory used for rewriting. When applying a tree filter, the command needs to temporarily check out the tree to some directory, which may consume considerable space in case of large projects. By default it does this in the *.git-rewrite/* directory but you can override that choice by this parameter.

-f

--force

git filter-branch refuses to start with an existing temporary directory or when there are already refs starting with *refs/original/*, unless forced.

<rev-list options>…

Arguments for *git rev-list*. All positive refs included by these options are rewritten. You may also specify options such as *--all*, but you must use -- to separate them from the *git filter-branch* options. Implies Remap to ancestor.

Remap to ancestor

By using rev-list[1] arguments, e.g., path limiters, you can limit the set of revisions which get rewritten. However, positive refs on the command line are distinguished: we don’t let them be excluded by such limiters. For this purpose, they are instead rewritten to point at the nearest ancestor that was not excluded.

Examples

Suppose you want to remove a file (containing confidential information or copyright violation) from all commits:

```
git filter-branch --tree-filter 'rm filename' HEAD
```

However, if the file is absent from the tree of some commit, a simple `rm filename` will fail for that tree and commit. Thus you may instead want to use `rm -f filename` as the script.

Using `--index-filter` with *git rm* yields a significantly faster version. Like with using `rm filename`, `git rm --cached filename` will fail if the file is absent from the tree of a commit. If you want to "completely forget" a file, it does not matter when it entered history, so we also add `--ignore-unmatch`:

```
git filter-branch --index-filter 'git rm --cached --ignore-unmatch filename' HEAD
```

Now, you will get the rewritten history saved in HEAD.

To rewrite the repository to look as if `foodir/` had been its project root, and discard all other history:

```
git filter-branch --subdirectory-filter foodir -- --all
```

Thus you can, e.g., turn a library subdirectory into a repository of its own. Note the `--` that separates *filter-branch* options from revision options, and the `--all` to rewrite all branches and tags.

To set a commit (which typically is at the tip of another history) to be the parent of the current initial commit, in order to paste the other history behind the current history:

```
git filter-branch --parent-filter 'sed "s/^\$/-p <graft-id>/"' HEAD
```

(if the parent string is empty - which happens when we are dealing with the initial commit - add graftcommit as a parent). Note that this assumes history with a single root (that is, no merge without common ancestors happened). If this is not the case, use:

```
git filter-branch --parent-filter \
    'test $GIT_COMMIT = <commit-id> && echo "-p <graft-id>" || cat' HEAD
```

or even simpler:

```
echo "$commit-id $graft-id" >> .git/info/grafts
git filter-branch $graft-id..HEAD
```

To remove commits authored by "Darl McBribe" from the history:

```
git filter-branch --commit-filter '
    if [ "$GIT_AUTHOR_NAME" = "Darl McBribe" ];
    then
        skip_commit "$@";
    else
        git commit-tree "$@";
    fi' HEAD
```

The function *skip_commit* is defined as follows:

```
skip_commit()
{
    shift;
    while [ -n "$1" ];
    do
        shift;
        map "$1";
        shift;
    done;
}
```

The shift magic first throws away the tree id and then the -p parameters. Note that this handles merges properly! In case Darl committed a merge between P1 and P2, it will be propagated properly and all children of the merge will become merge commits with P1,P2 as their parents instead of the merge commit.

NOTE the changes introduced by the commits, and which are not reverted by subsequent commits, will still be in the rewritten branch. If you want to throw out *changes* together with the commits, you should use the interactive mode of *git rebase*.

You can rewrite the commit log messages using `--msg-filter`. For example, *git svn-id* strings in a repository created by *git svn* can be removed this way:

```
git filter-branch --msg-filter '
    sed -e "/^git-svn-id:/d"
'
```

If you need to add *Acked-by* lines to, say, the last 10 commits (none of which is a merge), use this command:

```
git filter-branch --msg-filter '
    cat &&
    echo "Acked-by: Bugs Bunny <bunny@bugzilla.org>"
' HEAD~10..HEAD
```

The `--env-filter` option can be used to modify committer and/or author identity. For example, if you found out that your commits have the wrong identity due to a misconfigured user.email, you can make a correction, before publishing the project, like this:

```
git filter-branch --env-filter '
    if test "$GIT_AUTHOR_EMAIL" = "root@localhost"
    then
        GIT_AUTHOR_EMAIL=john@example.com
        export GIT_AUTHOR_EMAIL
    fi
    if test "$GIT_COMMITTER_EMAIL" = "root@localhost"
    then
        GIT_COMMITTER_EMAIL=john@example.com
        export GIT_COMMITTER_EMAIL
    fi
' -- --all
```

To restrict rewriting to only part of the history, specify a revision range in addition to the new branch name. The new branch name will point to the top-most revision that a *git rev-list* of this range will print.

Consider this history:

```
     D--E--F--G--H
    /     /
A--B-----C
```

To rewrite only commits D,E,F,G,H, but leave A, B and C alone, use:

```
git filter-branch ... C..H
```

To rewrite commits E,F,G,H, use one of these:

```
git filter-branch ... C..H --not D
git filter-branch ... D..H --not C
```

To move the whole tree into a subdirectory, or remove it from there:

```
git filter-branch --index-filter \
    'git ls-files -s | sed "s-\t\"*-&newsubdir/-" |
        GIT_INDEX_FILE=$GIT_INDEX_FILE.new \
            git update-index --index-info &&
     mv "$GIT_INDEX_FILE.new" "$GIT_INDEX_FILE"' HEAD
```

Checklist for Shrinking a Repository

git-filter-branch can be used to get rid of a subset of files, usually with some combination of `--index-filter` and `--subdirectory-filter`. People expect the resulting repository to be smaller than the original, but you need a few more steps to actually make it smaller, because Git tries hard not to lose your objects until you tell it to. First make sure that:

- You really removed all variants of a filename, if a blob was moved over its lifetime. `git log --name-only --follow --all -- filename` can help you find renames.
- You really filtered all refs: use `--tag-name-filter cat -- --all` when calling git-filter-branch.

Then there are two ways to get a smaller repository. A safer way is to clone, that keeps your original intact.

- Clone it with `git clone file:///path/to/repo`. The clone will not have the removed objects. See git-clone[1]. (Note that cloning with a plain path just hardlinks everything!)

If you really don't want to clone it, for whatever reasons, check the following points instead (in this order). This is a very destructive approach, so **make a backup** or go back to cloning it. You have been warned.

- Remove the original refs backed up by git-filter-branch: say `git for-each-ref --format="%(refname)" refs/original/ | xargs -n 1 git update-ref -d`.
- Expire all reflogs with `git reflog expire --expire=now --all`.
- Garbage collect all unreferenced objects with `git gc --prune=now` (or if your git-gc is not new enough to support arguments to `--prune`, use `git repack -ad; git prune` instead).

Notes

git-filter-branch allows you to make complex shell-scripted rewrites of your Git history, but you probably don't need this flexibility if you're simply *removing unwanted data* like large files or passwords. For those operations you may want to consider The BFG Repo-Cleaner,

a JVM-based alternative to git-filter-branch, typically at least 10-50x faster for those use-cases, and with quite different characteristics:

- Any particular version of a file is cleaned exactly *once*. The BFG, unlike git-filter-branch, does not give you the opportunity to handle a file differently based on where or when it was committed within your history. This constraint gives the core performance benefit of The BFG, and is well-suited to the task of cleansing bad data - you don't care *where* the bad data is, you just want it *gone*.
- By default The BFG takes full advantage of multi-core machines, cleansing commit file-trees in parallel. git-filter-branch cleans commits sequentially (i.e. in a single-threaded manner), though it *is* possible to write filters that include their own parallelism, in the scripts executed against each commit.
- The command options are much more restrictive than git-filter branch, and dedicated just to the tasks of removing unwanted data- e.g: `--strip-blobs-bigger-than 1M`.

GIT

Part of the git[1] suite

instaweb

NAME

git-instaweb - Instantly browse your working repository in gitweb

SYNOPSIS

```
git instaweb [--local] [--httpd=<httpd>] [--port=<port>]
               [--browser=<browser>]
git instaweb [--start] [--stop] [--restart]
```

DESCRIPTION

A simple script to set up `gitweb` and a web server for browsing the local repository.

OPTIONS

-l

--local

Only bind the web server to the local IP (127.0.0.1).

-d

--httpd

The HTTP daemon command-line that will be executed. Command-line options may be specified here, and the configuration file will be added at the end of the command-line. Currently apache2, lighttpd, mongoose, plackup and webrick are supported. (Default: lighttpd)

-m

--module-path

The module path (only needed if httpd is Apache). (Default: /usr/lib/apache2/modules)

-p

--port

The port number to bind the httpd to. (Default: 1234)

-b

--browser

The web browser that should be used to view the gitweb page. This will be passed to the *git web{litdd}browse* helper script along with the URL of the gitweb instance. See git-web{litdd}browse[1] for more information about this. If the script fails, the URL will be printed to stdout.

start

--start

Start the httpd instance and exit. Regenerate configuration files as necessary for spawning a new instance.

stop

--stop

Stop the httpd instance and exit. This does not generate any of the configuration files for spawning a new instance, nor does it close the browser.

restart

--restart

Restart the httpd instance and exit. Regenerate configuration files as necessary for spawning a new instance.

CONFIGURATION

You may specify configuration in your .git/config

```
[instaweb]
    local = true
    httpd = apache2 -f
    port = 4321
    browser = konqueror
    modulePath = /usr/lib/apache2/modules
```

If the configuration variable *instaweb.browser* is not set, *web.browser* will be used instead if it is defined. See git-web{litdd}browse[1] for more information about this.

SEE ALSO

gitweb[1]

GIT

Part of the git[1] suite

archive

NAME

git-archive - Create an archive of files from a named tree

SYNOPSIS

```
git archive [--format=<fmt>] [--list] [--prefix=<prefix>/] [<extra>]
          [-o <file> | --output=<file>] [--worktree-attributes]
          [--remote=<repo> [--exec=<git-upload-archive>]] <tree-ish>
          [<path>…]
```

DESCRIPTION

Creates an archive of the specified format containing the tree structure for the named tree, and writes it out to the standard output. If <prefix> is specified it is prepended to the filenames in the archive.

git archive behaves differently when given a tree ID versus when given a commit ID or tag ID. In the first case the current time is used as the modification time of each file in the archive. In the latter case the commit time as recorded in the referenced commit object is used instead. Additionally the commit ID is stored in a global extended pax header if the tar format is used; it can be extracted using *git get-tar-commit-id*. In ZIP files it is stored as a file comment.

OPTIONS

--format=<fmt>

Format of the resulting archive: *tar* or *zip*. If this option is not given, and the output file is specified, the format is inferred from the filename if possible (e.g. writing to "foo.zip" makes the output to be in the zip format). Otherwise the output format is `tar`.

-l

--list

Show all available formats.

-v

--verbose

Report progress to stderr.

--prefix=<prefix>/

Prepend <prefix>/ to each filename in the archive.

-o <file>

--output=<file>

Write the archive to <file> instead of stdout.

--worktree-attributes

Look for attributes in .gitattributes files in the working tree as well (see ATTRIBUTES).

<extra>

This can be any options that the archiver backend understands. See next section.

--remote=<repo>

Instead of making a tar archive from the local repository, retrieve a tar archive from a remote repository. Note that the remote repository may place restrictions on which sha1 expressions may be allowed in <tree-ish>. See git-upload-archive[1] for details.

--exec=<git-upload-archive>

Used with --remote to specify the path to the *git-upload-archive* on the remote side.

<tree-ish>

The tree or commit to produce an archive for.

<path>

Without an optional path parameter, all files and subdirectories of the current working directory are included in the archive. If one or more paths are specified, only these are included.

BACKEND EXTRA OPTIONS

zip

-0

Store the files instead of deflating them.

-9

Highest and slowest compression level. You can specify any number from 1 to 9 to adjust compression speed and ratio.

CONFIGURATION

tar.umask

This variable can be used to restrict the permission bits of tar archive entries. The default is 0002, which turns off the world write bit. The special value "user" indicates that the archiving user’s umask will be used instead. See umask(2) for details. If `--remote` is used then only the configuration of the remote repository takes effect.

tar.<format>.command

This variable specifies a shell command through which the tar output generated by `git archive` should be piped. The command is executed using the shell with the generated tar file on its standard input, and should produce the final output on its standard output. Any compression-level options will be passed to the command (e.g., "-9"). An output file with the same extension as `<format>` will be use this format if no other format is given.

The "tar.gz" and "tgz" formats are defined automatically and default to `gzip -cn`. You may override them with custom commands.

tar.<format>.remote

If true, enable `<format>` for use by remote clients via git-upload-archive[1]. Defaults to false for user-defined formats, but true for the "tar.gz" and "tgz" formats.

ATTRIBUTES

export-ignore

Files and directories with the attribute export-ignore won’t be added to archive files. See gitattributes[5] for details.

export-subst

If the attribute export-subst is set for a file then Git will expand several placeholders when adding this file to an archive. See gitattributes[5] for details.

Note that attributes are by default taken from the `.gitattributes` files in the tree that is being archived. If you want to tweak the way the output is generated after the fact (e.g. you committed without adding an appropriate export-ignore in its `.gitattributes`), adjust the checked out `.gitattributes` file as necessary and use `--worktree-attributes` option. Alternatively you can keep necessary attributes that should apply while archiving any tree in your `$GIT_DIR/info/attributes` file.

EXAMPLES

```
git archive --format=tar --prefix=junk/ HEAD | (cd /var/tmp/ && tar xf -)
```

Create a tar archive that contains the contents of the latest commit on the current branch, and extract it in the `/var/tmp/junk` directory.

```
git archive --format=tar --prefix=git-1.4.0/ v1.4.0 | gzip &gt;git-1.4.0.tar.gz
```

Create a compressed tarball for v1.4.0 release.

```
git archive --format=tar.gz --prefix=git-1.4.0/ v1.4.0 &gt;git-1.4.0.tar.gz
```

Same as above, but using the builtin tar.gz handling.

```
git archive --prefix=git-1.4.0/ -o git-1.4.0.tar.gz v1.4.0
```

Same as above, but the format is inferred from the output file.

```
git archive --format=tar --prefix=git-1.4.0/ v1.4.0^{tree} | gzip &gt;git-1.4.0.tar.gz
```

Create a compressed tarball for v1.4.0 release, but without a global extended pax header.

```
git archive --format=zip --prefix=git-docs/ HEAD:Documentation/ &gt; git-1.4.0-docs.zip
```

Put everything in the current head’s Documentation/ directory into *git-1.4.0-docs.zip*, with the prefix *git-docs/*.

```
git archive -o latest.zip HEAD
```

Create a Zip archive that contains the contents of the latest commit on the current branch. Note that the output format is inferred by the extension of the output file.

```
git config tar.tar.xz.command "xz -c"
```

Configure a "tar.xz" format for making LZMA-compressed tarfiles. You can use it specifying `--format=tar.xz` , or by creating an output file like `-o foo.tar.xz` .

SEE ALSO

gitattributes[5]

GIT

Part of the git[1] suite

bundle

NAME

git-bundle - Move objects and refs by archive

SYNOPSIS

```
git bundle create <file> <git-rev-list-args>
git bundle verify <file>
git bundle list-heads <file> [<refname>…]
git bundle unbundle <file> [<refname>…]
```

DESCRIPTION

Some workflows require that one or more branches of development on one machine be replicated on another machine, but the two machines cannot be directly connected, and therefore the interactive Git protocols (git, ssh, http) cannot be used. This command provides support for *git fetch* and *git pull* to operate by packaging objects and references in an archive at the originating machine, then importing those into another repository using *git fetch* and *git pull* after moving the archive by some means (e.g., by sneakernet). As no direct connection between the repositories exists, the user must specify a basis for the bundle that is held by the destination repository: the bundle assumes that all objects in the basis are already in the destination repository.

OPTIONS

create <file>

Used to create a bundle named *file*. This requires the *git-rev-list-args* arguments to define the bundle contents.

verify <file>

Used to check that a bundle file is valid and will apply cleanly to the current repository. This includes checks on the bundle format itself as well as checking that the prerequisite commits exist and are fully linked in the current repository. *git bundle* prints a list of missing commits, if any, and exits with a non-zero status.

list-heads <file>

Lists the references defined in the bundle. If followed by a list of references, only references matching those given are printed out.

unbundle <file>

Passes the objects in the bundle to *git index-pack* for storage in the repository, then prints the names of all defined references. If a list of references is given, only references matching those in the list are printed. This command is really plumbing, intended to be called only by *git fetch*.

<git-rev-list-args>

A list of arguments, acceptable to *git rev-parse* and *git rev-list* (and containing a named ref, see SPECIFYING REFERENCES below), that specifies the specific objects and references to transport. For example, `master~10..master` causes the current master reference to be packaged along with all objects added since its 10th ancestor commit. There is no explicit limit to the number of references and objects that may be packaged.

[<refname>…]

A list of references used to limit the references reported as available. This is principally of use to *git fetch*, which expects to receive only those references asked for and not necessarily everything in the pack (in this case, *git bundle* acts like *git fetch-pack*).

SPECIFYING REFERENCES

git bundle will only package references that are shown by *git show-ref*: this includes heads, tags, and remote heads. References such as `master~1` cannot be packaged, but are perfectly suitable for defining the basis. More than one reference may be packaged, and more than one basis can be specified. The objects packaged are those not contained in the union of the given bases. Each basis can be specified explicitly (e.g. `^master~10`), or implicitly (e.g. `master~10..master` , `--since=10.days.ago master`).

It is very important that the basis used be held by the destination. It is okay to err on the side of caution, causing the bundle file to contain objects already in the destination, as these are ignored when unpacking at the destination.

EXAMPLE

Assume you want to transfer the history from a repository R1 on machine A to another repository R2 on machine B. For whatever reason, direct connection between A and B is not allowed, but we can move data from A to B via some mechanism (CD, email, etc.). We want to update R2 with development made on the branch master in R1.

To bootstrap the process, you can first create a bundle that does not have any basis. You can use a tag to remember up to what commit you last processed, in order to make it easy to later update the other repository with an incremental bundle:

```
machineA$ cd R1
machineA$ git bundle create file.bundle master
machineA$ git tag -f lastR2bundle master
```

Then you transfer file.bundle to the target machine B. Because this bundle does not require any existing object to be extracted, you can create a new repository on machine B by cloning from it:

```
machineB$ git clone -b master /home/me/tmp/file.bundle R2
```

This will define a remote called "origin" in the resulting repository that lets you fetch and pull from the bundle. The $GIT_DIR/config file in R2 will have an entry like this:

```
[remote "origin"]
    url = /home/me/tmp/file.bundle
    fetch = refs/heads/*:refs/remotes/origin/*
```

To update the resulting mine.git repository, you can fetch or pull after replacing the bundle stored at /home/me/tmp/file.bundle with incremental updates.

After working some more in the original repository, you can create an incremental bundle to update the other repository:

```
machineA$ cd R1
machineA$ git bundle create file.bundle lastR2bundle..master
machineA$ git tag -f lastR2bundle master
```

You then transfer the bundle to the other machine to replace /home/me/tmp/file.bundle, and pull from it.

```
machineB$ cd R2
machineB$ git pull
```

If you know up to what commit the intended recipient repository should have the necessary objects, you can use that knowledge to specify the basis, giving a cut-off point to limit the revisions and objects that go in the resulting bundle. The previous example used the

lastR2bundle tag for this purpose, but you can use any other options that you would give to the git-log[1] command. Here are more examples:

You can use a tag that is present in both:

```
$ git bundle create mybundle v1.0.0..master
```

You can use a basis based on time:

```
$ git bundle create mybundle --since=10.days master
```

You can use the number of commits:

```
$ git bundle create mybundle -10 master
```

You can run `git-bundle verify` to see if you can extract from a bundle that was created with a basis:

```
$ git bundle verify mybundle
```

This will list what commits you must have in order to extract from the bundle and will error out if you do not have them.

A bundle from a recipient repository's point of view is just like a regular repository which it fetches or pulls from. You can, for example, map references when fetching:

```
$ git fetch mybundle master:localRef
```

You can also see what references it offers:

```
$ git ls-remote mybundle
```

GIT

Part of the git[1] suite

Server Admin

daemon

NAME

git-daemon - A really simple server for Git repositories

SYNOPSIS

```
git daemon [--verbose] [--syslog] [--export-all]
         [--timeout=<n>] [--init-timeout=<n>] [--max-connections=<n>]
         [--strict-paths] [--base-path=<path>] [--base-path-relaxed]
         [--user-path | --user-path=<path>]
         [--interpolated-path=<pathtemplate>]
         [--reuseaddr] [--detach] [--pid-file=<file>]
         [--enable=<service>] [--disable=<service>]
         [--allow-override=<service>] [--forbid-override=<service>]
         [--access-hook=<path>] [--[no-]informative-errors]
         [--inetd |
          [--listen=<host_or_ipaddr>] [--port=<n>]
          [--user=<user> [--group=<group>]]]
         [<directory>…]
```

DESCRIPTION

A really simple TCP Git daemon that normally listens on port "DEFAULT_GIT_PORT" aka 9418. It waits for a connection asking for a service, and will serve that service if it is enabled.

It verifies that the directory has the magic file "git-daemon-export-ok", and it will refuse to export any Git directory that hasn't explicitly been marked for export this way (unless the *--export-all* parameter is specified). If you pass some directory paths as *git daemon* arguments, you can further restrict the offers to a whitelist comprising of those.

By default, only `upload-pack` service is enabled, which serves *git fetch-pack* and *git ls-remote* clients, which are invoked from *git fetch*, *git pull*, and *git clone*.

This is ideally suited for read-only updates, i.e., pulling from Git repositories.

An `upload-archive` also exists to serve *git archive*.

OPTIONS

--strict-paths

Match paths exactly (i.e. don’t allow "/foo/repo" when the real path is "/foo/repo.git" or "/foo/repo/.git") and don’t do user-relative paths. *git daemon* will refuse to start when this option is enabled and no whitelist is specified.

--base-path=<path>

Remap all the path requests as relative to the given path. This is sort of "Git root" - if you run *git daemon* with *--base-path=/srv/git* on example.com, then if you later try to pull *git://example.com/hello.git*, *git daemon* will interpret the path as */srv/git/hello.git*.

--base-path-relaxed

If --base-path is enabled and repo lookup fails, with this option *git daemon* will attempt to lookup without prefixing the base path. This is useful for switching to --base-path usage, while still allowing the old paths.

--interpolated-path=<pathtemplate>

To support virtual hosting, an interpolated path template can be used to dynamically construct alternate paths. The template supports %H for the target hostname as supplied by the client but converted to all lowercase, %CH for the canonical hostname, %IP for the server’s IP address, %P for the port number, and %D for the absolute path of the named repository. After interpolation, the path is validated against the directory whitelist.

--export-all

Allow pulling from all directories that look like Git repositories (have the *objects* and *refs* subdirectories), even if they do not have the *git-daemon-export-ok* file.

--inetd

Have the server run as an inetd service. Implies --syslog. Incompatible with --detach, --port, --listen, --user and --group options.

--listen=<host_or_ipaddr>

Listen on a specific IP address or hostname. IP addresses can be either an IPv4 address or an IPv6 address if supported. If IPv6 is not supported, then --listen=hostname is also not supported and --listen must be given an IPv4 address. Can be given more than once. Incompatible with *--inetd* option.

--port=<n>

Listen on an alternative port. Incompatible with *--inetd* option.

--init-timeout=<n>

Timeout (in seconds) between the moment the connection is established and the client request is received (typically a rather low value, since that should be basically immediate).

--timeout=<n>

Timeout (in seconds) for specific client sub-requests. This includes the time it takes for the server to process the sub-request and the time spent waiting for the next client's request.

--max-connections=<n>

Maximum number of concurrent clients, defaults to 32. Set it to zero for no limit.

--syslog

Log to syslog instead of stderr. Note that this option does not imply --verbose, thus by default only error conditions will be logged.

--user-path

--user-path=<path>

Allow ~user notation to be used in requests. When specified with no parameter, requests to git://host/~alice/foo is taken as a request to access *foo* repository in the home directory of user `alice`. If `--user-path=path` is specified, the same request is taken as a request to access `path/foo` repository in the home directory of user `alice`.

--verbose

Log details about the incoming connections and requested files.

--reuseaddr

Use SO_REUSEADDR when binding the listening socket. This allows the server to restart without waiting for old connections to time out.

--detach

Detach from the shell. Implies --syslog.

--pid-file=<file>

Save the process id in *file*. Ignored when the daemon is run under `--inetd`.

--user=<user>

--group=<group>

Change daemon's uid and gid before entering the service loop. When only `--user` is given without `--group`, the primary group ID for the user is used. The values of the option are given to `getpwnam(3)` and `getgrnam(3)` and numeric IDs are not supported.

Giving these options is an error when used with `--inetd`; use the facility of inet daemon to achieve the same before spawning *git daemon* if needed.

Like many programs that switch user id, the daemon does not reset environment variables such as `$HOME` when it runs git programs, e.g. `upload-pack` and `receive-pack`. When using this option, you may also want to set and export `HOME` to point at the home directory of `<user>` before starting the daemon, and make sure any Git configuration files in that directory are readable by `<user>`.

--enable=<service>

--disable=<service>

Enable/disable the service site-wide per default. Note that a service disabled site-wide can still be enabled per repository if it is marked overridable and the repository enables the service with a configuration item.

--allow-override=<service>

--forbid-override=<service>

Allow/forbid overriding the site-wide default with per repository configuration. By default, all the services may be overridden.

--[no-]informative-errors

When informative errors are turned on, git-daemon will report more verbose errors to the client, differentiating conditions like "no such repository" from "repository not exported". This is more convenient for clients, but may leak information about the existence of unexported repositories. When informative errors are not enabled, all errors report "access denied" to the client. The default is --no-informative-errors.

--access-hook=<path>

Every time a client connects, first run an external command specified by the <path> with service name (e.g. "upload-pack"), path to the repository, hostname (%H), canonical hostname (%CH), IP address (%IP), and TCP port (%P) as its command-line arguments. The external command can decide to decline the service by exiting with a non-zero status (or to allow it by exiting with a zero status). It can also look at the $REMOTE_ADDR and $REMOTE_PORT environment variables to learn about the requestor when making this decision.

The external command can optionally write a single line to its standard output to be sent to the requestor as an error message when it declines the service.

<directory>

A directory to add to the whitelist of allowed directories. Unless --strict-paths is specified this will also include subdirectories of each named directory.

SERVICES

These services can be globally enabled/disabled using the command-line options of this command. If finer-grained control is desired (e.g. to allow *git archive* to be run against only in a few selected repositories the daemon serves), the per-repository configuration file can be used to enable or disable them.

upload-pack

This serves *git fetch-pack* and *git ls-remote* clients. It is enabled by default, but a repository can disable it by setting `daemon.uploadpack` configuration item to `false`.

upload-archive

This serves *git archive --remote*. It is disabled by default, but a repository can enable it by setting `daemon.uploadarch` configuration item to `true`.

receive-pack

This serves *git send-pack* clients, allowing anonymous push. It is disabled by default, as there is *no* authentication in the protocol (in other words, anybody can push anything into the repository, including removal of refs). This is solely meant for a closed LAN setting where everybody is friendly. This service can be enabled by setting `daemon.receivepack` configuration item to `true`.

EXAMPLES

We assume the following in /etc/services

```
$ grep 9418 /etc/services
git         9418/tcp        # Git Version Control System
```

git daemon as inetd server

To set up *git daemon* as an inetd service that handles any repository under the whitelisted set of directories, /pub/foo and /pub/bar, place an entry like the following into /etc/inetd all on one line:

```
    git stream tcp nowait nobody  /usr/bin/git
        git daemon --inetd --verbose --export-all
        /pub/foo /pub/bar
```

git daemon as inetd server for virtual hosts

To set up *git daemon* as an inetd service that handles repositories for different virtual hosts, `www.example.com` and `www.example.org`, place an entry like the following into `/etc/inetd` all on one line:

```
git stream tcp nowait nobody /usr/bin/git
    git daemon --inetd --verbose --export-all
    --interpolated-path=/pub/%H%D
    /pub/www.example.org/software
    /pub/www.example.com/software
    /software
```

In this example, the root-level directory `/pub` will contain a subdirectory for each virtual host name supported. Further, both hosts advertise repositories simply as `git://www.example.com/software/repo.git`. For pre-1.4.0 clients, a symlink from `/software` into the appropriate default repository could be made as well.

git daemon as regular daemon for virtual hosts

To set up *git daemon* as a regular, non-inetd service that handles repositories for multiple virtual hosts based on their IP addresses, start the daemon like this:

```
git daemon --verbose --export-all
    --interpolated-path=/pub/%IP/%D
    /pub/192.168.1.200/software
    /pub/10.10.220.23/software
```

In this example, the root-level directory `/pub` will contain a subdirectory for each virtual host IP address supported. Repositories can still be accessed by hostname though, assuming they correspond to these IP addresses.

selectively enable/disable services per repository

To enable *git archive --remote* and disable *git fetch* against a repository, have the following in the configuration file in the repository (that is the file *config* next to *HEAD*, *refs* and *objects*).

```
[daemon]
    uploadpack = false
    uploadarch = true
```

ENVIRONMENT

git daemon will set REMOTE_ADDR to the IP address of the client that connected to it, if the IP address is available. REMOTE_ADDR will be available in the environment of hooks called when services are performed.

GIT

Part of the git[1] suite

update-server-info

NAME

git-update-server-info - Update auxiliary info file to help dumb servers

SYNOPSIS

```
git update-server-info [--force]
```

DESCRIPTION

A dumb server that does not do on-the-fly pack generations must have some auxiliary information files in $GIT_DIR/info and $GIT_OBJECT_DIRECTORY/info directories to help clients discover what references and packs the server has. This command generates such auxiliary files.

OPTIONS

-f

--force

Update the info files from scratch.

OUTPUT

Currently the command updates the following files. Please see gitrepository-layout[5] for description of what they are for:

- objects/info/packs
- info/refs

GIT

Part of the git[1] suite

Plumbing Commands

cat-file

NAME

git-cat-file - Provide content or type and size information for repository objects

SYNOPSIS

```
git cat-file (-t [--allow-unknown-type]| -s [--allow-unknown-type]| -e | -p | <type> | --
git cat-file (--batch | --batch-check) [--follow-symlinks]
```

DESCRIPTION

In its first form, the command provides the content or the type of an object in the repository. The type is required unless *-t* or *-p* is used to find the object type, or *-s* is used to find the object size, or *--textconv* is used (which implies type "blob").

In the second form, a list of objects (separated by linefeeds) is provided on stdin, and the SHA-1, type, and size of each object is printed on stdout.

OPTIONS

<object>

The name of the object to show. For a more complete list of ways to spell object names, see the "SPECIFYING REVISIONS" section in gitrevisions[7].

-t

Instead of the content, show the object type identified by <object>.

-s

Instead of the content, show the object size identified by <object>.

-e

Suppress all output; instead exit with zero status if <object> exists and is a valid object.

-p

Pretty-print the contents of <object> based on its type.

<type>

Typically this matches the real type of <object> but asking for a type that can trivially be dereferenced from the given <object> is also permitted. An example is to ask for a "tree" with <object> being a commit object that contains it, or to ask for a "blob" with <object> being a tag object that points at it.

--textconv

Show the content as transformed by a textconv filter. In this case, <object> has be of the form <tree-ish>:<path>, or :<path> in order to apply the filter to the content recorded in the index at <path>.

--batch

--batch=<format>

Print object information and contents for each object provided on stdin. May not be combined with any other options or arguments. See the section `BATCH OUTPUT` below for details.

--batch-check

--batch-check=<format>

Print object information for each object provided on stdin. May not be combined with any other options or arguments. See the section `BATCH OUTPUT` below for details.

--batch-all-objects

Instead of reading a list of objects on stdin, perform the requested batch operation on all objects in the repository and any alternate object stores (not just reachable objects). Requires `--batch` or `--batch-check` be specified. Note that the objects are visited in order sorted by their hashes.

--buffer

Normally batch output is flushed after each object is output, so that a process can interactively read and write from `cat-file`. With this option, the output uses normal stdio buffering; this is much more efficient when invoking `--batch-check` on a large number of objects.

--allow-unknown-type

Allow -s or -t to query broken/corrupt objects of unknown type.

--follow-symlinks

With --batch or --batch-check, follow symlinks inside the repository when requesting objects with extended SHA-1 expressions of the form tree-ish:path-in-tree. Instead of providing output about the link itself, provide output about the linked-to object. If a symlink points outside the tree-ish (e.g. a link to /foo or a root-level link to ../foo), the portion of the link which is outside the tree will be printed.

This option does not (currently) work correctly when an object in the index is specified (e.g. `:link` instead of `HEAD:link`) rather than one in the tree.

This option cannot (currently) be used unless `--batch` or `--batch-check` is used.

For example, consider a git repository containing:

```
f: a file containing "hello\n"
link: a symlink to f
dir/link: a symlink to ../f
plink: a symlink to ../f
alink: a symlink to /etc/passwd
```

For a regular file `f`, `echo HEAD:f | git cat-file --batch` would print

```
ce013625030ba8dba906f756967f9e9ca394464a blob 6
```

And `echo HEAD:link | git cat-file --batch --follow-symlinks` would print the same thing, as would `HEAD:dir/link`, as they both point at `HEAD:f`.

Without `--follow-symlinks`, these would print data about the symlink itself. In the case of `HEAD:link`, you would see

```
4d1ae35ba2c8ec712fa2a379db44ad639ca277bd blob 1
```

Both `plink` and `alink` point outside the tree, so they would respectively print:

```
symlink 4
../f
```

```
symlink 11
/etc/passwd
```

OUTPUT

If *-t* is specified, one of the <type>.

If *-s* is specified, the size of the <object> in bytes.

If *-e* is specified, no output.

If *-p* is specified, the contents of <object> are pretty-printed.

If <type> is specified, the raw (though uncompressed) contents of the <object> will be returned.

BATCH OUTPUT

If `--batch` or `--batch-check` is given, `cat-file` will read objects from stdin, one per line, and print information about them. By default, the whole line is considered as an object, as if it were fed to git-rev-parse[1].

You can specify the information shown for each object by using a custom `<format>`. The `<format>` is copied literally to stdout for each object, with placeholders of the form `%(atom)` expanded, followed by a newline. The available atoms are:

`objectname`

The 40-hex object name of the object.

`objecttype`

The type of of the object (the same as `cat-file -t` reports).

`objectsize`

The size, in bytes, of the object (the same as `cat-file -s` reports).

`objectsize:disk`

The size, in bytes, that the object takes up on disk. See the note about on-disk sizes in the `CAVEATS` section below.

`deltabase`

If the object is stored as a delta on-disk, this expands to the 40-hex sha1 of the delta base object. Otherwise, expands to the null sha1 (40 zeroes). See `CAVEATS` below.

`rest`

If this atom is used in the output string, input lines are split at the first whitespace boundary. All characters before that whitespace are considered to be the object name; characters after that first run of whitespace (i.e., the "rest" of the line) are output in place of the `%(rest)` atom.

If no format is specified, the default format is `%(objectname) %(objecttype) %(objectsize)`.

If `--batch` is specified, the object information is followed by the object contents (consisting of `%(objectsize)` bytes), followed by a newline.

For example, `--batch` without a custom format would produce:

```
<sha1> SP <type> SP <size> LF
<contents> LF
```

Whereas `--batch-check='%(objectname) %(objecttype)'` would produce:

```
<sha1> SP <type> LF
```

If a name is specified on stdin that cannot be resolved to an object in the repository, then `cat-file` will ignore any custom format and print:

```
<object> SP missing LF
```

If --follow-symlinks is used, and a symlink in the repository points outside the repository, then `cat-file` will ignore any custom format and print:

```
symlink SP <size> LF
<symlink> LF
```

The symlink will either be absolute (beginning with a /), or relative to the tree root. For instance, if dir/link points to ../../foo, then <symlink> will be ../foo. <size> is the size of the symlink in bytes.

If --follow-symlinks is used, the following error messages will be displayed:

```
<object> SP missing LF
```

is printed when the initial symlink requested does not exist.

```
dangling SP <size> LF
<object> LF
```

is printed when the initial symlink exists, but something that it (transitive-of) points to does not.

```
loop SP <size> LF
<object> LF
```

is printed for symlink loops (or any symlinks that require more than 40 link resolutions to resolve).

```
notdir SP <size> LF
<object> LF
```

is printed when, during symlink resolution, a file is used as a directory name.

CAVEATS

Note that the sizes of objects on disk are reported accurately, but care should be taken in drawing conclusions about which refs or objects are responsible for disk usage. The size of a packed non-delta object may be much larger than the size of objects which delta against it, but the choice of which object is the base and which is the delta is arbitrary and is subject to change during a repack.

Note also that multiple copies of an object may be present in the object database; in this case, it is undefined which copy’s size or delta base will be reported.

GIT

Part of the git[1] suite

commit-tree

NAME

git-commit-tree - Create a new commit object

SYNOPSIS

```
git commit-tree <tree> [(-p <parent>)…]
git commit-tree [(-p <parent>)…] [-S[<keyid>]] [(-m <message>)…]
          [(-F <file>)…] <tree>
```

DESCRIPTION

This is usually not what an end user wants to run directly. See git-commit[1] instead.

Creates a new commit object based on the provided tree object and emits the new commit object id on stdout. The log message is read from the standard input, unless `-m` or `-F` options are given.

A commit object may have any number of parents. With exactly one parent, it is an ordinary commit. Having more than one parent makes the commit a merge between several lines of history. Initial (root) commits have no parents.

While a tree represents a particular directory state of a working directory, a commit represents that state in "time", and explains how to get there.

Normally a commit would identify a new "HEAD" state, and while Git doesn’t care where you save the note about that state, in practice we tend to just write the result to the file that is pointed at by `.git/HEAD`, so that we can always see what the last committed state was.

OPTIONS

<tree>

An existing tree object

-p <parent>

Each *-p* indicates the id of a parent commit object.

-m <message>

A paragraph in the commit log message. This can be given more than once and each <message> becomes its own paragraph.

-F <file>

Read the commit log message from the given file. Use `-` to read from the standard input.

-S[<keyid>]

--gpg-sign[=<keyid>]

GPG-sign commits. The `keyid` argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a space.

--no-gpg-sign

Countermand `commit.gpgSign` configuration variable that is set to force each and every commit to be signed.

Commit Information

A commit encapsulates:

- all parent object ids
- author name, email and date
- committer name and email and the commit time.

While parent object ids are provided on the command line, author and committer information is taken from the following environment variables, if set:

```
GIT_AUTHOR_NAME
GIT_AUTHOR_EMAIL
GIT_AUTHOR_DATE
GIT_COMMITTER_NAME
GIT_COMMITTER_EMAIL
GIT_COMMITTER_DATE
```

(nb "<", ">" and "\n"s are stripped)

In case (some of) these environment variables are not set, the information is taken from the configuration items user.name and user.email, or, if not present, the environment variable EMAIL, or, if that is not set, system user name and the hostname used for outgoing mail (taken from `/etc/mailname` and falling back to the fully qualified hostname when that file does not exist).

A commit comment is read from stdin. If a changelog entry is not provided via "<" redirection, *git commit-tree* will just wait for one to be entered and terminated with ^D.

DATE FORMATS

The GIT_AUTHOR_DATE, GIT_COMMITTER_DATE environment variables support the following date formats:

Git internal format

It is `<unix timestamp> <time zone offset>`, where `<unix timestamp>` is the number of seconds since the UNIX epoch. `<time zone offset>` is a positive or negative offset from UTC. For example CET (which is 2 hours ahead UTC) is `+0200`.

RFC 2822

The standard email format as described by RFC 2822, for example `Thu, 07 Apr 2005 22:13:13 +0200`.

ISO 8601

Time and date specified by the ISO 8601 standard, for example `2005-04-07T22:13:13`. The parser accepts a space instead of the `T` character as well.

> Note
>
> In addition, the date part is accepted in the following formats: `YYYY.MM.DD`, `MM/DD/YYYY` and `DD.MM.YYYY`.

Discussion

Git is to some extent character encoding agnostic.

- The contents of the blob objects are uninterpreted sequences of bytes. There is no encoding translation at the core level.

- Path names are encoded in UTF-8 normalization form C. This applies to tree objects, the index file, ref names, as well as path names in command line arguments, environment variables and config files (`.git/config` (see git-config[1]), gitignore[5], gitattributes[5] and gitmodules[5]).

  Note that Git at the core level treats path names simply as sequences of non-NUL bytes, there are no path name encoding conversions (except on Mac and Windows). Therefore, using non-ASCII path names will mostly work even on platforms and file systems that use legacy extended ASCII encodings. However, repositories created on

such systems will not work properly on UTF-8-based systems (e.g. Linux, Mac, Windows) and vice versa. Additionally, many Git-based tools simply assume path names to be UTF-8 and will fail to display other encodings correctly.

- Commit log messages are typically encoded in UTF-8, but other extended ASCII encodings are also supported. This includes ISO-8859-x, CP125x and many others, but *not* UTF-16/32, EBCDIC and CJK multi-byte encodings (GBK, Shift-JIS, Big5, EUC-x, CP9xx etc.).

Although we encourage that the commit log messages are encoded in UTF-8, both the core and Git Porcelain are designed not to force UTF-8 on projects. If all participants of a particular project find it more convenient to use legacy encodings, Git does not forbid it. However, there are a few things to keep in mind.

1. *git commit* and *git commit-tree* issues a warning if the commit log message given to it does not look like a valid UTF-8 string, unless you explicitly say your project uses a legacy encoding. The way to say this is to have i18n.commitencoding in `.git/config` file, like this:

   ```
   [i18n]
       commitencoding = ISO-8859-1
   ```

   Commit objects created with the above setting record the value of `i18n.commitencoding` in its `encoding` header. This is to help other people who look at them later. Lack of this header implies that the commit log message is encoded in UTF-8.

2. *git log*, *git show*, *git blame* and friends look at the `encoding` header of a commit object, and try to re-code the log message into UTF-8 unless otherwise specified. You can specify the desired output encoding with `i18n.logoutputencoding` in `.git/config` file, like this:

   ```
   [i18n]
       logoutputencoding = ISO-8859-1
   ```

   If you do not have this configuration variable, the value of `i18n.commitencoding` is used instead.

Note that we deliberately chose not to re-code the commit log message when a commit is made to force UTF-8 at the commit object level, because re-coding to UTF-8 is not necessarily a reversible operation.

FILES

/etc/mailname

SEE ALSO

git-write-tree[1]

GIT

Part of the git[1] suite

count-objects

NAME

git-count-objects - Count unpacked number of objects and their disk consumption

SYNOPSIS

```
git count-objects [-v] [-H | --human-readable]
```

DESCRIPTION

This counts the number of unpacked object files and disk space consumed by them, to help you decide when it is a good time to repack.

OPTIONS

-v

--verbose

Report in more detail:

count: the number of loose objects

size: disk space consumed by loose objects, in KiB (unless -H is specified)

in-pack: the number of in-pack objects

size-pack: disk space consumed by the packs, in KiB (unless -H is specified)

prune-packable: the number of loose objects that are also present in the packs. These objects could be pruned using `git prune-packed`.

garbage: the number of files in object database that are neither valid loose objects nor valid packs

size-garbage: disk space consumed by garbage files, in KiB (unless -H is specified)

-H

--human-readable

Print sizes in human readable format

GIT

Part of the git[1] suite

diff-index

NAME

git-diff-index - Compare a tree to the working tree or index

SYNOPSIS

```
git diff-index [-m] [--cached] [<common diff options>] <tree-ish> [<path>…]
```

DESCRIPTION

Compares the content and mode of the blobs found in a tree object with the corresponding tracked files in the working tree, or with the corresponding paths in the index. When <path> arguments are present, compares only paths matching those patterns. Otherwise all tracked files are compared.

OPTIONS

-p

-u

--patch

Generate patch (see section on generating patches).

-s

--no-patch

Suppress diff output. Useful for commands like `git show` that show the patch by default, or to cancel the effect of `--patch`.

-U<n>

--unified=<n>

Generate diffs with <n> lines of context instead of the usual three. Implies `-p`.

--raw

Generate the diff in raw format. This is the default.

--patch-with-raw

Synonym for `-p --raw`.

--minimal

Spend extra time to make sure the smallest possible diff is produced.

--patience

Generate a diff using the "patience diff" algorithm.

--histogram

Generate a diff using the "histogram diff" algorithm.

--diff-algorithm={patience|minimal|histogram|myers}

Choose a diff algorithm. The variants are as follows:

`default`, `myers`

The basic greedy diff algorithm. Currently, this is the default.

`minimal`

Spend extra time to make sure the smallest possible diff is produced.

`patience`

Use "patience diff" algorithm when generating patches.

`histogram`

This algorithm extends the patience algorithm to "support low-occurrence common elements".

For instance, if you configured diff.algorithm variable to a non-default value and want to use the default one, then you have to use `--diff-algorithm=default` option.

--stat[=<width>[,<name-width>[,<count>]]]

Generate a diffstat. By default, as much space as necessary will be used for the filename part, and the rest for the graph part. Maximum width defaults to terminal width, or 80 columns if not connected to a terminal, and can be overridden by `<width>`. The width of the filename part can be limited by giving another width `<name-width>` after a comma. The width of the graph part can be limited by using `--stat-graph-width=<width>` (affects all commands generating a stat graph) or by

setting `diff.statGraphWidth=<width>` (does not affect `git format-patch`). By giving a third parameter `<count>`, you can limit the output to the first `<count>` lines, followed by `...` if there are more.

These parameters can also be set individually with `--stat-width=<width>`, `--stat-name-width=<name-width>` and `--stat-count=<count>`.

--numstat

Similar to `--stat`, but shows number of added and deleted lines in decimal notation and pathname without abbreviation, to make it more machine friendly. For binary files, outputs two `-` instead of saying `0 0`.

--shortstat

Output only the last line of the `--stat` format containing total number of modified files, as well as number of added and deleted lines.

--dirstat[=<param1,param2,…>]

Output the distribution of relative amount of changes for each sub-directory. The behavior of `--dirstat` can be customized by passing it a comma separated list of parameters. The defaults are controlled by the `diff.dirstat` configuration variable (see git-config[1]). The following parameters are available:

`changes`

Compute the dirstat numbers by counting the lines that have been removed from the source, or added to the destination. This ignores the amount of pure code movements within a file. In other words, rearranging lines in a file is not counted as much as other changes. This is the default behavior when no parameter is given.

`lines`

Compute the dirstat numbers by doing the regular line-based diff analysis, and summing the removed/added line counts. (For binary files, count 64-byte chunks instead, since binary files have no natural concept of lines). This is a more expensive `--dirstat` behavior than the `changes` behavior, but it does count rearranged lines within a file as much as other changes. The resulting output is consistent with what you get from the other `--*stat` options.

`files`

Compute the dirstat numbers by counting the number of files changed. Each changed file counts equally in the dirstat analysis. This is the computationally cheapest `--dirstat` behavior, since it does not have to look at the file contents at all.

`cumulative`

Count changes in a child directory for the parent directory as well. Note that when using `cumulative`, the sum of the percentages reported may exceed 100%. The default (non-cumulative) behavior can be specified with the `noncumulative` parameter.

<limit>

An integer parameter specifies a cut-off percent (3% by default). Directories contributing less than this percentage of the changes are not shown in the output.

Example: The following will count changed files, while ignoring directories with less than 10% of the total amount of changed files, and accumulating child directory counts in the parent directories: `--dirstat=files,10,cumulative`.

--summary

Output a condensed summary of extended header information such as creations, renames and mode changes.

--patch-with-stat

Synonym for `-p --stat`.

-z

When `--raw`, `--numstat`, `--name-only` or `--name-status` has been given, do not munge pathnames and use NULs as output field terminators.

Without this option, each pathname output will have TAB, LF, double quotes, and backslash characters replaced with `\t`, `\n`, `\"`, and `\\`, respectively, and the pathname will be enclosed in double quotes if any of those replacements occurred.

--name-only

Show only names of changed files.

--name-status

Show only names and status of changed files. See the description of the `--diff-filter` option on what the status letters mean.

--submodule[=<format>]

Specify how differences in submodules are shown. When `--submodule` or `--submodule=log` is given, the *log* format is used. This format lists the commits in the range like git-submodule[1] `summary` does. Omitting the `--submodule` option or specifying `--submodule=short`, uses the *short* format. This format just shows the names of the commits at the beginning and end of the range. Can be tweaked via the `diff.submodule` configuration variable.

--color[=<when>]

Show colored diff. `--color` (i.e. without =*<when>*) is the same as `--color=always`. *<when>* can be one of `always`, `never`, or `auto`.

--no-color

Turn off colored diff. It is the same as `--color=never`.

--word-diff[=<mode>]

Show a word diff, using the <mode> to delimit changed words. By default, words are delimited by whitespace; see `--word-diff-regex` below. The <mode> defaults to *plain*, and must be one of:

color

Highlight changed words using only colors. Implies `--color`.

plain

Show words as `[-removed-]` and `{+added+}`. Makes no attempts to escape the delimiters if they appear in the input, so the output may be ambiguous.

porcelain

Use a special line-based format intended for script consumption. Added/removed/unchanged runs are printed in the usual unified diff format, starting with a `+` / `-` / `` ` character at the beginning of the line and extending to the end of the line. Newl ``
~` on a line of its own.

none

Disable word diff again.

Note that despite the name of the first mode, color is used to highlight the changed parts in all modes if enabled.

--word-diff-regex=<regex>

Use <regex> to decide what a word is, instead of considering runs of non-whitespace to be a word. Also implies `--word-diff` unless it was already enabled.

Every non-overlapping match of the <regex> is considered a word. Anything between these matches is considered whitespace and ignored(!) for the purposes of finding differences. You may want to append `|[^[:space:]]` to your regular expression to make sure that it matches all non-whitespace characters. A match that contains a newline is silently truncated(!) at the newline.

For example, `--word-diff-regex=.` will treat each character as a word and, correspondingly, show differences character by character.

The regex can also be set via a diff driver or configuration option, see gitattributes[1] or git-config[1]. Giving it explicitly overrides any diff driver or configuration setting. Diff drivers override configuration settings.

--color-words[=<regex>]

Equivalent to `--word-diff=color` plus (if a regex was specified) `--word-diff-regex=<regex>`.

--no-renames

Turn off rename detection, even when the configuration file gives the default to do so.

--check

Warn if changes introduce whitespace errors. What are considered whitespace errors is controlled by `core.whitespace` configuration. By default, trailing whitespaces (including lines that solely consist of whitespaces) and a space character that is immediately followed by a tab character inside the initial indent of the line are considered whitespace errors. Exits with non-zero status if problems are found. Not compatible with --exit-code.

--ws-error-highlight=<kind>

Highlight whitespace errors on lines specified by <kind> in the color specified by `color.diff.whitespace`. <kind> is a comma separated list of `old`, `new`, `context`. When this option is not given, only whitespace errors in `new` lines are highlighted. E.g. `--ws-error-highlight=new,old` highlights whitespace errors on both deleted and added lines. `all` can be used as a short-hand for `old,new,context`.

--full-index

Instead of the first handful of characters, show the full pre- and post-image blob object names on the "index" line when generating patch format output.

--binary

In addition to `--full-index`, output a binary diff that can be applied with `git-apply`.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object name in diff-raw format output and diff-tree header lines, show only a partial prefix. This is independent of the `--full-index` option above, which controls the diff-patch output format. Non default number of digits can be specified with `--abbrev=<n>`.

-B[<n>][/<m>]

--break-rewrites[=[<n>][/<m>]]

Break complete rewrite changes into pairs of delete and create. This serves two purposes:

It affects the way a change that amounts to a total rewrite of a file not as a series of deletion and insertion mixed together with a very few lines that happen to match textually as the context, but as a single deletion of everything old followed by a single insertion of everything new, and the number `m` controls this aspect of the -B option (defaults to 60%). `-B/70%` specifies that less than 30% of the original should remain in the result for Git to consider it a total rewrite (i.e. otherwise the resulting patch will be a series of deletion and insertion mixed together with context lines).

When used with -M, a totally-rewritten file is also considered as the source of a rename (usually -M only considers a file that disappeared as the source of a rename), and the number `n` controls this aspect of the -B option (defaults to 50%). `-B20%` specifies that a change with addition and deletion compared to 20% or more of the file's size are eligible for being picked up as a possible source of a rename to another file.

-M[<n>]

--find-renames[=<n>]

Detect renames. If `n` is specified, it is a threshold on the similarity index (i.e. amount of addition/deletions compared to the file's size). For example, `-M90%` means Git should consider a delete/add pair to be a rename if more than 90% of the file hasn't changed. Without a `%` sign, the number is to be read as a fraction, with a decimal point before it. I.e., `-M5` becomes 0.5, and is thus the same as `-M50%`. Similarly, `-M05` is the same as `-M5%`. To limit detection to exact renames, use `-M100%`. The default similarity index is 50%.

-C[<n>]

--find-copies[=<n>]

Detect copies as well as renames. See also `--find-copies-harder`. If `n` is specified, it has the same meaning as for `-M<n>`.

--find-copies-harder

For performance reasons, by default, `-C` option finds copies only if the original file of the copy was modified in the same changeset. This flag makes the command inspect unmodified files as candidates for the source of copy. This is a very expensive operation for large projects, so use it with caution. Giving more than one `-C` option has the same effect.

-D

--irreversible-delete

Omit the preimage for deletes, i.e. print only the header but not the diff between the preimage and `/dev/null`. The resulting patch is not meant to be applied with `patch` or `git apply`; this is solely for people who want to just concentrate on reviewing the text after the change. In addition, the output obviously lack enough information to apply such a patch in reverse, even manually, hence the name of the option.

When used together with `-B`, omit also the preimage in the deletion part of a delete/create pair.

-l<num>

The `-M` and `-C` options require O(n^2) processing time where n is the number of potential rename/copy targets. This option prevents rename/copy detection from running if the number of rename/copy targets exceeds the specified number.

--diff-filter=[(A|C|D|M|R|T|U|X|B)…[*]]

Select only files that are Added (`A`), Copied (`C`), Deleted (`D`), Modified (`M`), Renamed (`R`), have their type (i.e. regular file, symlink, submodule, …) changed (`T`), are Unmerged (`U`), are Unknown (`X`), or have had their pairing Broken (`B`). Any combination of the filter characters (including none) can be used. When `*` (All-or-none) is added to the combination, all paths are selected if there is any file that matches other criteria in the comparison; if there is no file that matches other criteria, nothing is selected.

-S<string>

Look for differences that change the number of occurrences of the specified string (i.e. addition/deletion) in a file. Intended for the scripter’s use.

It is useful when you’re looking for an exact block of code (like a struct), and want to know the history of that block since it first came into being: use the feature iteratively to feed the interesting block in the preimage back into `-S`, and keep going until you get the very first version of the block.

-G<regex>

Look for differences whose patch text contains added/removed lines that match <regex>.

To illustrate the difference between `-S<regex> --pickaxe-regex` and `-G<regex>`, consider a commit with the following diff in the same file:

```
+    return !regexec(regexp, two-&gt;ptr, 1, &regmatch, 0);
...
-    hit = !regexec(regexp, mf2.ptr, 1, &regmatch, 0);
```

While `git log -G"regexec\(regexp"` will show this commit, `git log -S"regexec\(regexp" --pickaxe-regex` will not (because the number of occurrences of that string did not change).

See the *pickaxe* entry in gitdiffcore[7] for more information.

--pickaxe-all

When `-S` or `-G` finds a change, show all the changes in that changeset, not just the files that contain the change in <string>.

--pickaxe-regex

Treat the <string> given to `-S` as an extended POSIX regular expression to match.

-O<orderfile>

Output the patch in the order specified in the <orderfile>, which has one shell glob pattern per line. This overrides the `diff.orderFile` configuration variable (see git-config[1]). To cancel `diff.orderFile`, use `-O/dev/null`.

-R

Swap two inputs; that is, show differences from index or on-disk file to tree contents.

--relative[=<path>]

When run from a subdirectory of the project, it can be told to exclude changes outside the directory and show pathnames relative to it with this option. When you are not in a subdirectory (e.g. in a bare repository), you can name which subdirectory to make the output relative to by giving a <path> as an argument.

-a

--text

Treat all files as text.

--ignore-space-at-eol

Ignore changes in whitespace at EOL.

-b

--ignore-space-change

Ignore changes in amount of whitespace. This ignores whitespace at line end, and considers all other sequences of one or more whitespace characters to be equivalent.

-w

--ignore-all-space

Ignore whitespace when comparing lines. This ignores differences even if one line has whitespace where the other line has none.

--ignore-blank-lines

Ignore changes whose lines are all blank.

--inter-hunk-context=<lines>

Show the context between diff hunks, up to the specified number of lines, thereby fusing hunks that are close to each other.

-W

--function-context

Show whole surrounding functions of changes.

--exit-code

Make the program exit with codes similar to diff(1). That is, it exits with 1 if there were differences and 0 means no differences.

--quiet

Disable all output of the program. Implies `--exit-code`.

--ext-diff

Allow an external diff helper to be executed. If you set an external diff driver with gitattributes[5], you need to use this option with git-log[1] and friends.

--no-ext-diff

Disallow external diff drivers.

--textconv

--no-textconv

Allow (or disallow) external text conversion filters to be run when comparing binary files. See gitattributes[5] for details. Because textconv filters are typically a one-way conversion, the resulting diff is suitable for human consumption, but cannot be applied. For this reason, textconv filters are enabled by default only for git-diff[1] and git-log[1], but not for git-format-patch[1] or diff plumbing commands.

--ignore-submodules[=<when>]

Ignore changes to submodules in the diff generation. <when> can be either "none", "untracked", "dirty" or "all", which is the default. Using "none" will consider the submodule modified when it either contains untracked or modified files or its HEAD differs from the commit recorded in the superproject and can be used to override any settings of the *ignore* option in git-config[1] or gitmodules[5]. When "untracked" is used submodules are not considered dirty when they only contain untracked content (but they are still scanned for modified content). Using "dirty" ignores all changes to the work tree of submodules, only changes to the commits stored in the superproject are shown (this was the behavior until 1.7.0). Using "all" hides all changes to submodules.

--src-prefix=<prefix>

Show the given source prefix instead of "a/".

--dst-prefix=<prefix>

Show the given destination prefix instead of "b/".

--no-prefix

Do not show any source or destination prefix.

For more detailed explanation on these common options, see also gitdiffcore[7].

<tree-ish>

The id of a tree object to diff against.

--cached

do not consider the on-disk file at all

-m

By default, files recorded in the index but not checked out are reported as deleted. This flag makes *git diff-index* say that all non-checked-out files are up to date.

Raw output format

The raw output format from "git-diff-index", "git-diff-tree", "git-diff-files" and "git diff --raw" are very similar.

These commands all compare two sets of things; what is compared differs:

git-diff-index <tree-ish>

compares the <tree-ish> and the files on the filesystem.

git-diff-index --cached <tree-ish>

compares the <tree-ish> and the index.

git-diff-tree [-r] <tree-ish-1> <tree-ish-2> [<pattern>…]

compares the trees named by the two arguments.

git-diff-files [<pattern>…]

compares the index and the files on the filesystem.

The "git-diff-tree" command begins its output by printing the hash of what is being compared. After that, all the commands print one output line per changed file.

An output line is formatted this way:

```
in-place edit  :100644 100644 bcd1234... 0123456... M file0
copy-edit      :100644 100644 abcd123... 1234567... C68 file1 file2
rename-edit    :100644 100644 abcd123... 1234567... R86 file1 file3
create         :000000 100644 0000000... 1234567... A file4
delete         :100644 000000 1234567... 0000000... D file5
unmerged       :000000 000000 0000000... 0000000... U file6
```

That is, from the left to the right:

1. a colon.
2. mode for "src"; 000000 if creation or unmerged.
3. a space.
4. mode for "dst"; 000000 if deletion or unmerged.
5. a space.
6. sha1 for "src"; 0{40} if creation or unmerged.
7. a space.
8. sha1 for "dst"; 0{40} if creation, unmerged or "look at work tree".
9. a space.
10. status, followed by optional "score" number.
11. a tab or a NUL when *-z* option is used.
12. path for "src"
13. a tab or a NUL when *-z* option is used; only exists for C or R.
14. path for "dst"; only exists for C or R.

15. an LF or a NUL when *-z* option is used, to terminate the record.

Possible status letters are:

- A: addition of a file
- C: copy of a file into a new one
- D: deletion of a file
- M: modification of the contents or mode of a file
- R: renaming of a file
- T: change in the type of the file
- U: file is unmerged (you must complete the merge before it can be committed)
- X: "unknown" change type (most probably a bug, please report it)

Status letters C and R are always followed by a score (denoting the percentage of similarity between the source and target of the move or copy). Status letter M may be followed by a score (denoting the percentage of dissimilarity) for file rewrites.

<sha1> is shown as all 0’s if a file is new on the filesystem and it is out of sync with the index.

Example:

```
:100644 100644 5be4a4...... 000000...... M file.c
```

When `-z` option is not used, TAB, LF, and backslash characters in pathnames are represented as `\t`, `\n`, and `\\`, respectively.

diff format for merges

"git-diff-tree", "git-diff-files" and "git-diff --raw" can take *-c* or *--cc* option to generate diff output also for merge commits. The output differs from the format described above in the following way:

1. there is a colon for each parent
2. there are more "src" modes and "src" sha1
3. status is concatenated status characters for each parent
4. no optional "score" number

5. single path, only for "dst"

Example:

```
::100644 100644 100644 fabadb8... cc95eb0... 4866510... MM	describe.c
```

Note that *combined diff* lists only files which were modified from all parents.

Generating patches with -p

When "git-diff-index", "git-diff-tree", or "git-diff-files" are run with a *-p* option, "git diff" without the *--raw* option, or "git log" with the "-p" option, they do not produce the output described above; instead they produce a patch file. You can customize the creation of such patches via the GIT_EXTERNAL_DIFF and the GIT_DIFF_OPTS environment variables.

What the -p option produces is slightly different from the traditional diff format:

1. It is preceded with a "git diff" header that looks like this:

   ```
   diff --git a/file1 b/file2
   ```

   The `a/` and `b/` filenames are the same unless rename/copy is involved. Especially, even for a creation or a deletion, `/dev/null` is *not* used in place of the `a/` or `b/` filenames.

   When rename/copy is involved, `file1` and `file2` show the name of the source file of the rename/copy and the name of the file that rename/copy produces, respectively.

2. It is followed by one or more extended header lines:

   ```
   old mode &lt;mode&gt;
   new mode &lt;mode&gt;
   deleted file mode &lt;mode&gt;
   new file mode &lt;mode&gt;
   copy from &lt;path&gt;
   copy to &lt;path&gt;
   rename from &lt;path&gt;
   rename to &lt;path&gt;
   similarity index &lt;number&gt;
   dissimilarity index &lt;number&gt;
   index &lt;hash&gt;..&lt;hash&gt; &lt;mode&gt;
   ```

   File modes are printed as 6-digit octal numbers including the file type and file permission bits.

   Path names in extended headers do not include the `a/` and `b/` prefixes.

The similarity index is the percentage of unchanged lines, and the dissimilarity index is the percentage of changed lines. It is a rounded down integer, followed by a percent sign. The similarity index value of 100% is thus reserved for two equal files, while 100% dissimilarity means that no line from the old file made it into the new one.

The index line includes the SHA-1 checksum before and after the change. The <mode> is included if the file mode does not change; otherwise, separate lines indicate the old and the new mode.

3. TAB, LF, double quote and backslash characters in pathnames are represented as `\t`, `\n`, `\"` and `\\`, respectively. If there is need for such substitution then the whole pathname is put in double quotes.

4. All the `file1` files in the output refer to files before the commit, and all the `file2` files refer to files after the commit. It is incorrect to apply each change to each file sequentially. For example, this patch will swap a and b:

```
diff --git a/a b/b
rename from a
rename to b
diff --git a/b b/a
rename from b
rename to a
```

combined diff format

Any diff-generating command can take the `-c` or `--cc` option to produce a *combined diff* when showing a merge. This is the default format when showing merges with git-diff[1] or git-show[1]. Note also that you can give the `-m` option to any of these commands to force generation of diffs with individual parents of a merge.

A *combined diff* format looks like this:

```
diff --combined describe.c
index fabadb8,cc95eb0..4866510
--- a/describe.c
+++ b/describe.c
@@@ -98,20 -98,12 +98,20 @@@
    return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
  }

- static void describe(char *arg)
 -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
  {
 +    unsigned char sha1[20];
 +    struct commit *cmit;
    struct commit_list *list;
    static int initialized = 0;
    struct commit_name *n;

 +    if (get_sha1(arg, sha1) < 0)
 +        usage(describe_usage);
 +    cmit = lookup_commit_reference(sha1);
 +    if (!cmit)
 +        usage(describe_usage);
 +
    if (!initialized) {
        initialized = 1;
        for_each_ref(get_name);
```

1. It is preceded with a "git diff" header, that looks like this (when *-c* option is used):

   ```
   diff --combined file
   ```

   or like this (when *--cc* option is used):

   ```
   diff --cc file
   ```

2. It is followed by one or more extended header lines (this example shows a merge with two parents):

   ```
   index &lt;hash&gt;;,&lt;hash&gt;;..&lt;hash&gt;;
   mode &lt;mode&gt;;,&lt;mode&gt;;..&lt;mode&gt;;
   new file mode &lt;mode&gt;;
   deleted file mode &lt;mode&gt;;,&lt;mode&gt;;
   ```

   The `mode <mode>;,<mode>;..<mode>;` line appears only if at least one of the <mode> is different from the rest. Extended headers with information about detected contents movement (renames and copying detection) are designed to work with diff of two <tree-ish> and are not used by combined diff format.

3. It is followed by two-line from-file/to-file header

   ```
   --- a/file
   +++ b/file
   ```

Similar to two-line header for traditional *unified* diff format, `/dev/null` is used to signal created or deleted files.

4. Chunk header format is modified to prevent people from accidentally feeding it to `patch -p1`. Combined diff format was created for review of merge commit changes, and was not meant for apply. The change is similar to the change in the extended *index* header:

```
@@@ &lt;from-file-range&gt; &lt;from-file-range&gt; &lt;to-file-range&gt; @@@
```

There are (number of parents + 1) `@` characters in the chunk header for combined diff format.

Unlike the traditional *unified* diff format, which shows two files A and B with a single column that has `-` (minus — appears in A but removed in B), `+` (plus — missing in A but added to B), or `" "` (space — unchanged) prefix, this format compares two or more files file1, file2,… with one file X, and shows how X differs from each of fileN. One column for each of fileN is prepended to the output line to note how X's line is different from it.

A `-` character in the column N means that the line appears in fileN but it does not appear in the result. A `+` character in the column N means that the line appears in the result, and fileN does not have that line (in other words, the line was added, from the point of view of that parent).

In the above example output, the function signature was changed from both files (hence two `-` removals from both file1 and file2, plus `++` to mean one line that was added does not appear in either file1 or file2). Also eight other lines are the same from file1 but do not appear in file2 (hence prefixed with `+`).

When shown by `git diff-tree -c`, it compares the parents of a merge commit with the merge result (i.e. file1..fileN are the parents). When shown by `git diff-files -c`, it compares the two unresolved merge parents with the working tree file (i.e. file1 is stage 2 aka "our version", file2 is stage 3 aka "their version").

other diff formats

The `--summary` option describes newly added, deleted, renamed and copied files. The `--stat` option adds diffstat(1) graph to the output. These options can be combined with other options, such as `-p`, and are meant for human consumption.

When showing a change that involves a rename or a copy, `--stat` output formats the pathnames compactly by combining common prefix and suffix of the pathnames. For example, a change that moves `arch/i386/Makefile` to `arch/x86/Makefile` while modifying 4

lines will be shown like this:

```
arch/{i386 => x86}/Makefile    |   4 +--
```

The `--numstat` option gives the diffstat(1) information but is designed for easier machine consumption. An entry in `--numstat` output looks like this:

```
1    2    README
3    1    arch/{i386 => x86}/Makefile
```

That is, from left to right:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. pathname (possibly with rename/copy information);
6. a newline.

When `-z` output option is in effect, the output is formatted this way:

```
1    2    README NUL
3    1    NUL arch/i386/Makefile NUL arch/x86/Makefile NUL
```

That is:

1. the number of added lines;
2. a tab;
3. the number of deleted lines;
4. a tab;
5. a NUL (only exists if renamed/copied);
6. pathname in preimage;
7. a NUL (only exists if renamed/copied);
8. pathname in postimage (only exists if renamed/copied);
9. a NUL.

The extra `NUL` before the preimage path in renamed case is to allow scripts that read the output to tell if the current record being read is a single-path record or a rename/copy record without reading ahead. After reading added and deleted lines, reading up to `NUL` would yield the pathname, but if that is `NUL`, the record will show two paths.

Operating Modes

You can choose whether you want to trust the index file entirely (using the *--cached* flag) or ask the diff logic to show any files that don't match the stat state as being "tentatively changed". Both of these operations are very useful indeed.

Cached Mode

If *--cached* is specified, it allows you to ask:

```
show me the differences between HEAD and the current index
contents (the ones I'd write using 'git write-tree')
```

For example, let's say that you have worked on your working directory, updated some files in the index and are ready to commit. You want to see exactly **what** you are going to commit, without having to write a new tree object and compare it that way, and to do that, you just do

```
git diff-index --cached HEAD
```

Example: let's say I had renamed `commit.c` to `git-commit.c`, and I had done an `update-index` to make that effective in the index file. `git diff-files` wouldn't show anything at all, since the index file matches my working directory. But doing a *git diff-index* does:

```
torvalds@ppc970:~/git> git diff-index --cached HEAD
-100644 blob    4161aecc6700a2eb579e842af0b7f22b98443f74        commit.c
+100644 blob    4161aecc6700a2eb579e842af0b7f22b98443f74        git-commit.c
```

You can see easily that the above is a rename.

In fact, `git diff-index --cached` **should** always be entirely equivalent to actually doing a *git write-tree* and comparing that. Except this one is much nicer for the case where you just want to check where you are.

So doing a `git diff-index --cached` is basically very useful when you are asking yourself "what have I already marked for being committed, and what's the difference to a previous tree".

Non-cached Mode

The "non-cached" mode takes a different approach, and is potentially the more useful of the two in that what it does can't be emulated with a *git write-tree* + *git diff-tree*. Thus that's the default mode. The non-cached version asks the question:

```
show me the differences between HEAD and the currently checked out
tree - index contents _and_ files that aren't up-to-date
```

which is obviously a very useful question too, since that tells you what you **could** commit. Again, the output matches the *git diff-tree -r* output to a tee, but with a twist.

The twist is that if some file doesn't match the index, we don't have a backing store thing for it, and we use the magic "all-zero" sha1 to show that. So let's say that you have edited `kernel/sched.c`, but have not actually done a *git update-index* on it yet - there is no "object" associated with the new state, and you get:

```
torvalds@ppc970:~/v2.6/linux> git diff-index --abbrev HEAD
:100644 100664 7476bb... 000000...      kernel/sched.c
```

i.e., it shows that the tree has changed, and that `kernel/sched.c` has is not up-to-date and may contain new stuff. The all-zero sha1 means that to get the real diff, you need to look at the object in the working directory directly rather than do an object-to-object diff.

> Note
>
> As with other commands of this type, *git diff-index* does not actually look at the contents of the file at all. So maybe `kernel/sched.c` hasn't actually changed, and it's just that you touched it. In either case, it's a note that you need to *git update-index* it to make the index be in sync.
>
> Note
>
> You can have a mixture of files show up as "has been updated" and "is still dirty in the working directory" together. You can always tell which file is in which state, since the "has been updated" ones show a valid sha1, and the "not in sync with the index" ones will always have the special all-zero sha1.

GIT

Part of the git[1] suite

for-each-ref

NAME

git-for-each-ref - Output information on each ref

SYNOPSIS

```
git for-each-ref [--count=<count>] [--shell|--perl|--python|--tcl]
           [(--sort=<key>)…] [--format=<format>] [<pattern>…]
           [--points-at <object>] [(--merged | --no-merged) [<object>]]
           [--contains [<object>]]
```

DESCRIPTION

Iterate over all refs that match `<pattern>` and show them according to the given `<format>`, after sorting them according to the given set of `<key>`. If `<count>` is given, stop after showing that many refs. The interpolated values in `<format>` can optionally be quoted as string literals in the specified host language allowing their direct evaluation in that language.

OPTIONS

<count>

By default the command shows all refs that match `<pattern>`. This option makes it stop after showing that many refs.

<key>

A field name to sort on. Prefix `-` to sort in descending order of the value. When unspecified, `refname` is used. You may use the --sort=<key> option multiple times, in which case the last key becomes the primary key.

<format>

A string that interpolates `%(fieldname)` from the object pointed at by a ref being shown. If `fieldname` is prefixed with an asterisk (`*`) and the ref points at a tag object, the value for the field in the object tag refers is used. When unspecified, defaults to

`%(objectname) SPC %(objecttype) TAB %(refname)`. It also interpolates `%%` to `%`, and `%xx` where `xx` are hex digits interpolates to character with hex code `xx`; for example `%00` interpolates to `\0` (NUL), `%09` to `\t` (TAB) and `%0a` to `\n` (LF).

<pattern>…

If one or more patterns are given, only refs are shown that match against at least one pattern, either using fnmatch(3) or literally, in the latter case matching completely or from the beginning up to a slash.

--shell

--perl

--python

--tcl

If given, strings that substitute `%(fieldname)` placeholders are quoted as string literals suitable for the specified host language. This is meant to produce a scriptlet that can directly be `eval`ed.

--points-at <object>

Only list refs which points at the given object.

--merged [<object>]

Only list refs whose tips are reachable from the specified commit (HEAD if not specified).

--no-merged [<object>]

Only list refs whose tips are not reachable from the specified commit (HEAD if not specified).

--contains [<object>]

Only list tags which contain the specified commit (HEAD if not specified).

FIELD NAMES

Various values from structured fields in referenced objects can be used to interpolate into the resulting output, or as sort keys.

For all objects, the following names can be used:

refname

The name of the ref (the part after $GIT_DIR/). For a non-ambiguous short name of the ref append `:short`. The option core.warnAmbiguousRefs is used to select the strict abbreviation mode. If `strip=<N>` is appended, strips `<N>` slash-separated path components from the front of the refname (e.g., `%(refname:strip=2)` turns `refs/tags/foo` into `foo`. `<N>` must be a positive integer. If a displayed ref has fewer components than `<N>`, the command aborts with an error.

objecttype

The type of the object (`blob` , `tree` , `commit` , `tag`).

objectsize

The size of the object (the same as *git cat-file -s* reports).

objectname

The object name (aka SHA-1). For a non-ambiguous abbreviation of the object name append `:short`.

upstream

The name of a local ref which can be considered “upstream” from the displayed ref. Respects `:short` in the same way as `refname` above. Additionally respects `:track` to show "[ahead N, behind M]" and `:trackshort` to show the terse version: ">" (ahead), "<" (behind), "<>" (ahead and behind), or "=" (in sync). Has no effect if the ref does not have tracking information associated with it.

push

The name of a local ref which represents the `@{push}` location for the displayed ref. Respects `:short` , `:track` , and `:trackshort` options as `upstream` does. Produces an empty string if no `@{push}` ref is configured.

HEAD

* if HEAD matches current ref (the checked out branch), ' ' otherwise.

color

Change output color. Followed by `:<colorname>`, where names are described in `color.branch.*`.

align

Left-, middle-, or right-align the content between %(align:…) and %(end). The "align:" is followed by `width=<width>` and `position=<position>` in any order separated by a comma, where the `<position>` is either left, right or middle, default being left and

`<width>` is the total length of the content with alignment. For brevity, the "width=" and/or "position=" prefixes may be omitted, and bare <width> and <position> used instead. For instance, `%(align:<width>,<position>)`. If the contents length is more than the width then no alignment is performed. If used with *--quote* everything in between %(align:…) and %(end) is quoted, but if nested then only the topmost level performs quoting.

In addition to the above, for commit and tag objects, the header field names (`tree` , `parent` , `object` , `type` , and `tag`) can be used to specify the value in the header field.

For commit and tag objects, the special `creatordate` and `creator` fields will correspond to the appropriate date or name-email-date tuple from the `committer` or `tagger` fields depending on the object type. These are intended for working on a mix of annotated and lightweight tags.

Fields that have name-email-date tuple as its value (`author` , `committer` , and `tagger`) can be suffixed with `name` , `email` , and `date` to extract the named component.

The complete message in a commit and tag object is `contents` . Its first line is `contents:subject` , where subject is the concatenation of all lines of the commit message up to the first blank line. The next line is *contents:body*, where body is all of the lines after the first blank line. The optional GPG signature is `contents:signature` . The first `N` lines of the message is obtained using `contents:lines=N` .

For sorting purposes, fields with numeric values sort in numeric order (`objectsize` , `authordate` , `committerdate` , `creatordate` , `taggerdate`). All other fields are used to sort in their byte-value order.

There is also an option to sort by versions, this can be done by using the fieldname `version:refname` or its alias `v:refname` .

In any case, a field name that refers to a field inapplicable to the object referred by the ref does not cause an error. It returns an empty string instead.

As a special case for the date-type fields, you may specify a format for the date by adding `:` followed by date format name (see the values the `--date` option to :git-rev-list[1] takes).

EXAMPLES

An example directly producing formatted text. Show the most recent 3 tagged commits:

```
#!/bin/sh

git for-each-ref --count=3 --sort='-*authordate' \
--format='From: %(*authorname) %(*authoremail)
Subject: %(*subject)
Date: %(*authordate)
Ref: %(*refname)

%(*body)
' 'refs/tags'
```

A simple example showing the use of shell eval on the output, demonstrating the use of --shell. List the prefixes of all heads:

```
#!/bin/sh

git for-each-ref --shell --format="ref=%(refname)" refs/heads | \
while read entry
do
    eval "$entry"
    echo `dirname $ref`
done
```

A bit more elaborate report on tags, demonstrating that the format may be an entire script:

```
#!/bin/sh

fmt='
    r=%(refname)
    t=%(*objecttype)
    T=${r#refs/tags/}

    o=%(*objectname)
    n=%(*authorname)
    e=%(*authoremail)
    s=%(*subject)
    d=%(*authordate)
    b=%(*body)

    kind=Tag
    if test "z$t" = z
    then
        # could be a lightweight tag
        t=%(objecttype)
        kind="Lightweight tag"
        o=%(objectname)
        n=%(authorname)
        e=%(authoremail)
        s=%(subject)
        d=%(authordate)
        b=%(body)
    fi
    echo "$kind $T points at a $t object $o"
    if test "z$t" = zcommit
    then
        echo "The commit was authored by $n $e
at $d, and titled

    $s

Its message reads as:
"
        echo "$b" | sed -e "s/^/    /"
        echo
    fi
'

eval=`git for-each-ref --shell --format="$fmt" \
    --sort='*objecttype' \
    --sort=-taggerdate \
    refs/tags`
eval "$eval"
```

SEE ALSO

git-show-ref[1]

GIT

Part of the git[1] suite

hash-object

NAME

git-hash-object - Compute object ID and optionally creates a blob from a file

SYNOPSIS

```
git hash-object [-t <type>] [-w] [--path=<file>|--no-filters] [--stdin [--literally]] [--
git hash-object [-t <type>] [-w] --stdin-paths [--no-filters]
```

DESCRIPTION

Computes the object ID value for an object with specified type with the contents of the named file (which can be outside of the work tree), and optionally writes the resulting object into the object database. Reports its object ID to its standard output. This is used by *git cvsimport* to update the index without modifying files in the work tree. When <type> is not specified, it defaults to "blob".

OPTIONS

-t <type>

Specify the type (default: "blob").

-w

Actually write the object into the object database.

--stdin

Read the object from standard input instead of from a file.

--stdin-paths

Read file names from the standard input, one per line, instead of from the command-line.

--path

Hash object as it were located at the given path. The location of file does not directly influence on the hash value, but path is used to determine what Git filters should be applied to the object before it can be placed to the object database, and, as result of applying filters, the actual blob put into the object database may differ from the given file. This option is mainly useful for hashing temporary files located outside of the working directory or files read from stdin.

--no-filters

Hash the contents as is, ignoring any input filter that would have been chosen by the attributes mechanism, including the end-of-line conversion. If the file is read from standard input then this is always implied, unless the `--path` option is given.

--literally

Allow `--stdin` to hash any garbage into a loose object which might not otherwise pass standard object parsing or git-fsck checks. Useful for stress-testing Git itself or reproducing characteristics of corrupt or bogus objects encountered in the wild.

GIT

Part of the git[1] suite

ls-files

NAME

git-ls-files - Show information about files in the index and the working tree

SYNOPSIS

```
git ls-files [-z] [-t] [-v]
        (--[cached|deleted|others|ignored|stage|unmerged|killed|modified])*
        (-[c|d|o|i|s|u|k|m])*
        [--eol]
        [-x <pattern>|--exclude=<pattern>]
        [-X <file>|--exclude-from=<file>]
        [--exclude-per-directory=<file>]
        [--exclude-standard]
        [--error-unmatch] [--with-tree=<tree-ish>]
        [--full-name] [--abbrev] [--] [<file>…]
```

DESCRIPTION

This merges the file listing in the directory cache index with the actual working directory list, and shows different combinations of the two.

One or more of the options below may be used to determine the files shown:

OPTIONS

-c

--cached

Show cached files in the output (default)

-d

--deleted

Show deleted files in the output

-m

--modified

Show modified files in the output

-o

--others

Show other (i.e. untracked) files in the output

-i

--ignored

Show only ignored files in the output. When showing files in the index, print only those matched by an exclude pattern. When showing "other" files, show only those matched by an exclude pattern.

-s

--stage

Show staged contents' object name, mode bits and stage number in the output.

--directory

If a whole directory is classified as "other", show just its name (with a trailing slash) and not its whole contents.

--no-empty-directory

Do not list empty directories. Has no effect without --directory.

-u

--unmerged

Show unmerged files in the output (forces --stage)

-k

--killed

Show files on the filesystem that need to be removed due to file/directory conflicts for checkout-index to succeed.

-z

\0 line termination on output.

-x <pattern>

--exclude=<pattern>

Skip untracked files matching pattern. Note that pattern is a shell wildcard pattern. See EXCLUDE PATTERNS below for more information.

-X <file>

--exclude-from=<file>

Read exclude patterns from <file>; 1 per line.

--exclude-per-directory=<file>

Read additional exclude patterns that apply only to the directory and its subdirectories in <file>.

--exclude-standard

Add the standard Git exclusions: .git/info/exclude, .gitignore in each directory, and the user's global exclusion file.

--error-unmatch

If any <file> does not appear in the index, treat this as an error (return 1).

--with-tree=<tree-ish>

When using --error-unmatch to expand the user supplied <file> (i.e. path pattern) arguments to paths, pretend that paths which were removed in the index since the named <tree-ish> are still present. Using this option with `-s` or `-u` options does not make any sense.

-t

This feature is semi-deprecated. For scripting purpose, git-status[1] `--porcelain` and git-diff-files[1] `--name-status` are almost always superior alternatives, and users should look at git-status[1] `--short` or git-diff[1] `--name-status` for more user-friendly alternatives.

This option identifies the file status with the following tags (followed by a space) at the start of each line:

H

cached

S

skip-worktree

M

unmerged

R

removed/deleted

C

modified/changed

K

to be killed

?

other

-v

Similar to `-t`, but use lowercase letters for files that are marked as *assume unchanged* (see git-update-index[1]).

--full-name

When run from a subdirectory, the command usually outputs paths relative to the current directory. This option forces paths to be output relative to the project top directory.

--abbrev[=<n>]

Instead of showing the full 40-byte hexadecimal object lines, show only a partial prefix. Non default number of digits can be specified with --abbrev=<n>.

--debug

After each line that describes a file, add more data about its cache entry. This is intended to show as much information as possible for manual inspection; the exact format may change at any time.

--eol

Show <eolinfo> and <eolattr> of files. <eolinfo> is the file content identification used by Git when the "text" attribute is "auto" (or not set and core.autocrlf is not false). <eolinfo> is either "-text", "none", "lf", "crlf", "mixed" or "".

"" means the file is not a regular file, it is not in the index or not accessible in the working tree.

<eolattr> is the attribute that is used when checking out or committing, it is either "", "-text", "text", "text=auto", "text eol=lf", "text eol=crlf". Note: Currently Git does not support "text=auto eol=lf" or "text=auto eol=crlf", that may change in the future.

Both the <eolinfo> in the index ("i/<eolinfo>") and in the working tree ("w/<eolinfo>") are shown for regular files, followed by the ("attr/<eolattr>").

--

Do not interpret any more arguments as options.

<file>

Files to show. If no files are given all files which match the other specified criteria are shown.

Output

git ls-files just outputs the filenames unless *--stage* is specified in which case it outputs:

```
[<tag> ]<mode> <object> <stage> <file>
```

git ls-files --eol will show i/<eolinfo><SPACES>w/<eolinfo><SPACES>attr/<eolattr><SPACE*><TAB><file>

git ls-files --unmerged and *git ls-files --stage* can be used to examine detailed information on unmerged paths.

For an unmerged path, instead of recording a single mode/SHA-1 pair, the index records up to three such pairs; one from tree O in stage 1, A in stage 2, and B in stage 3. This information can be used by the user (or the porcelain) to see what should eventually be recorded at the path. (see git-read-tree[1] for more information on state)

When `-z` option is not used, TAB, LF, and backslash characters in pathnames are represented as `\t` , `\n` , and `\\` , respectively.

Exclude Patterns

git ls-files can use a list of "exclude patterns" when traversing the directory tree and finding files to show when the flags --others or --ignored are specified. gitignore[5] specifies the format of exclude patterns.

These exclude patterns come from these places, in order:

1. The command-line flag --exclude=<pattern> specifies a single pattern. Patterns are ordered in the same order they appear in the command line.

2. The command-line flag --exclude-from=<file> specifies a file containing a list of patterns. Patterns are ordered in the same order they appear in the file.

3. The command-line flag --exclude-per-directory=<name> specifies a name of the file in each directory *git ls-files* examines, normally `.gitignore`. Files in deeper directories take precedence. Patterns are ordered in the same order they appear in the files.

A pattern specified on the command line with --exclude or read from the file specified with --exclude-from is relative to the top of the directory tree. A pattern read from a file specified by --exclude-per-directory is relative to the directory that the pattern file appears in.

SEE ALSO

git-read-tree[1], gitignore[5]

GIT

Part of the git[1] suite

merge-base

NAME

git-merge-base - Find as good common ancestors as possible for a merge

SYNOPSIS

```
git merge-base [-a|--all] <commit> <commit>…
git merge-base [-a|--all] --octopus <commit>…
git merge-base --is-ancestor <commit> <commit>
git merge-base --independent <commit>…
git merge-base --fork-point <ref> [<commit>]
```

DESCRIPTION

git merge-base finds best common ancestor(s) between two commits to use in a three-way merge. One common ancestor is *better* than another common ancestor if the latter is an ancestor of the former. A common ancestor that does not have any better common ancestor is a *best common ancestor*, i.e. a *merge base*. Note that there can be more than one merge base for a pair of commits.

OPERATION MODES

As the most common special case, specifying only two commits on the command line means computing the merge base between the given two commits.

More generally, among the two commits to compute the merge base from, one is specified by the first commit argument on the command line; the other commit is a (possibly hypothetical) commit that is a merge across all the remaining commits on the command line.

As a consequence, the *merge base* is not necessarily contained in each of the commit arguments if more than two commits are specified. This is different from git-show-branch[1] when used with the `--merge-base` option.

--octopus

Compute the best common ancestors of all supplied commits, in preparation for an n-way merge. This mimics the behavior of *git show-branch --merge-base*.

--independent

Instead of printing merge bases, print a minimal subset of the supplied commits with the same ancestors. In other words, among the commits given, list those which cannot be reached from any other. This mimics the behavior of *git show-branch --independent*.

--is-ancestor

Check if the first <commit> is an ancestor of the second <commit>, and exit with status 0 if true, or with status 1 if not. Errors are signaled by a non-zero status that is not 1.

--fork-point

Find the point at which a branch (or any history that leads to <commit>) forked from another branch (or any reference) <ref>. This does not just look for the common ancestor of the two commits, but also takes into account the reflog of <ref> to see if the history leading to <commit> forked from an earlier incarnation of the branch <ref> (see discussion on this mode below).

OPTIONS

-a

--all

Output all merge bases for the commits, instead of just one.

DISCUSSION

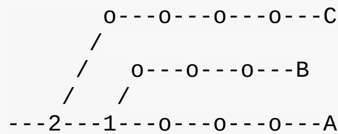
Given two commits *A* and *B*, `git merge-base A B` will output a commit which is reachable from both *A* and *B* through the parent relationship.

For example, with this topology:

```
     o---o---o---B
    /
---o---1---o---o---o---A
```

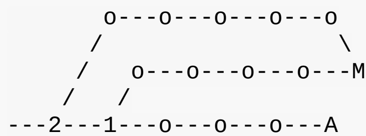
the merge base between *A* and *B* is *1*.

Given three commits *A*, *B* and *C*, `git merge-base A B C` will compute the merge base between *A* and a hypothetical commit *M*, which is a merge between *B* and *C*. For example, with this topology:



```
         o---o---o---o---C
        /
       /   o---o---o---B
      /   /
  ---2---1---o---o---o---A
```

the result of `git merge-base A B C` is *1*. This is because the equivalent topology with a merge commit *M* between *B* and *C* is:

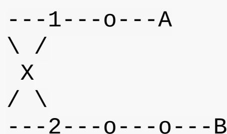


```
         o---o---o---o---o
        /                 \
       /   o---o---o---o---M
      /   /
  ---2---1---o---o---o---A
```

and the result of `git merge-base A M` is *1*. Commit *2* is also a common ancestor between *A* and *M*, but *1* is a better common ancestor, because *2* is an ancestor of *1*. Hence, *2* is not a merge base.

The result of `git merge-base --octopus A B C` is *2*, because *2* is the best common ancestor of all commits.

When the history involves criss-cross merges, there can be more than one *best* common ancestor for two commits. For example, with this topology:



```
  ---1---o---A
  \ /
   X
  / \
  ---2---o---o---B
```

both *1* and *2* are merge-bases of A and B. Neither one is better than the other (both are *best* merge bases). When the `--all` option is not given, it is unspecified which best one is output.

A common idiom to check "fast-forward-ness" between two commits A and B is (or at least used to be) to compute the merge base between A and B, and check if it is the same as A, in which case, A is an ancestor of B. You will see this idiom used often in older scripts.

```
A=$(git rev-parse --verify A)
if test "$A" = "$(git merge-base A B)"
then
    ... A is an ancestor of B ...
fi
```

In modern git, you can say this in a more direct way:

```
if git merge-base --is-ancestor A B
then
    ... A is an ancestor of B ...
fi
```

instead.

Discussion on fork-point mode

After working on the `topic` branch created with `git checkout -b topic origin/master`, the history of remote-tracking branch `origin/master` may have been rewound and rebuilt, leading to a history of this shape:

```
         o---B1
        /
---o---o---B2--o---o---o---B (origin/master)
    \
     B3
      \
       Derived (topic)
```

where `origin/master` used to point at commits B3, B2, B1 and now it points at B, and your `topic` branch was started on top of it back when `origin/master` was at B3. This mode uses the reflog of `origin/master` to find B3 as the fork point, so that the `topic` can be rebased on top of the updated `origin/master` by:

```
$ fork_point=$(git merge-base --fork-point origin/master topic)
$ git rebase --onto origin/master $fork_point topic
```

See also

git-rev-list[1], git-show-branch[1], git-merge[1]

GIT

Part of the git[1] suite

read-tree

NAME

git-read-tree - Reads tree information into the index

SYNOPSIS

```
git read-tree [[-m [--trivial] [--aggressive] | --reset | --prefix=<prefix>]
        [-u [--exclude-per-directory=<gitignore>] | -i]]
        [--index-output=<file>] [--no-sparse-checkout]
        (--empty | <tree-ish1> [<tree-ish2> [<tree-ish3>]])
```

DESCRIPTION

Reads the tree information given by <tree-ish> into the index, but does not actually **update** any of the files it "caches". (see: git-checkout-index[1])

Optionally, it can merge a tree into the index, perform a fast-forward (i.e. 2-way) merge, or a 3-way merge, with the `-m` flag. When used with `-m`, the `-u` flag causes it to also update the files in the work tree with the result of the merge.

Trivial merges are done by *git read-tree* itself. Only conflicting paths will be in unmerged state when *git read-tree* returns.

OPTIONS

-m

Perform a merge, not just a read. The command will refuse to run if your index file has unmerged entries, indicating that you have not finished previous merge you started.

--reset

Same as -m, except that unmerged entries are discarded instead of failing.

-u

After a successful merge, update the files in the work tree with the result of the merge.

-i

Usually a merge requires the index file as well as the files in the working tree to be up to date with the current head commit, in order not to lose local changes. This flag disables the check with the working tree and is meant to be used when creating a merge of trees that are not directly related to the current working tree status into a temporary index file.

-n

--dry-run

Check if the command would error out, without updating the index or the files in the working tree for real.

-v

Show the progress of checking files out.

--trivial

Restrict three-way merge by *git read-tree* to happen only if there is no file-level merging required, instead of resolving merge for trivial cases and leaving conflicting files unresolved in the index.

--aggressive

Usually a three-way merge by *git read-tree* resolves the merge for really trivial cases and leaves other cases unresolved in the index, so that porcelains can implement different merge policies. This flag makes the command resolve a few more cases internally:

- when one side removes a path and the other side leaves the path unmodified. The resolution is to remove that path.
- when both sides remove a path. The resolution is to remove that path.
- when both sides add a path identically. The resolution is to add that path.

--prefix=<prefix>/

Keep the current index contents, and read the contents of the named tree-ish under the directory at `<prefix>`. The command will refuse to overwrite entries that already existed in the original index file. Note that the `<prefix>/` value must end with a slash.

--exclude-per-directory=<gitignore>

When running the command with `-u` and `-m` options, the merge result may need to overwrite paths that are not tracked in the current branch. The command usually refuses to proceed with the merge to avoid losing such a path. However this safety valve sometimes gets in the way. For example, it often happens that the other branch added a file that used to be a generated file in your branch, and the safety valve triggers when you try to switch to

that branch after you ran `make` but before running `make clean` to remove the generated file. This option tells the command to read per-directory exclude file (usually *.gitignore*) and allows such an untracked but explicitly ignored file to be overwritten.

--index-output=<file>

Instead of writing the results out to `$GIT_INDEX_FILE`, write the resulting index in the named file. While the command is operating, the original index file is locked with the same mechanism as usual. The file must allow to be rename(2)ed into from a temporary file that is created next to the usual index file; typically this means it needs to be on the same filesystem as the index file itself, and you need write permission to the directories the index file and index output file are located in.

--no-sparse-checkout

Disable sparse checkout support even if `core.sparseCheckout` is true.

--empty

Instead of reading tree object(s) into the index, just empty it.

<tree-ish#>

The id of the tree object(s) to be read/merged.

Merging

If `-m` is specified, *git read-tree* can perform 3 kinds of merge, a single tree merge if only 1 tree is given, a fast-forward merge with 2 trees, or a 3-way merge if 3 trees are provided.

Single Tree Merge

If only 1 tree is specified, *git read-tree* operates as if the user did not specify `-m`, except that if the original index has an entry for a given pathname, and the contents of the path match with the tree being read, the stat info from the index is used. (In other words, the index's stat()s take precedence over the merged tree's).

That means that if you do a `git read-tree -m <newtree>` followed by a `git checkout-index -f -u -a`, the *git checkout-index* only checks out the stuff that really changed.

This is used to avoid unnecessary false hits when *git diff-files* is run after *git read-tree*.

Two Tree Merge

Typically, this is invoked as `git read-tree -m $H $M`, where $H is the head commit of the current repository, and $M is the head of a foreign tree, which is simply ahead of $H (i.e. we are in a fast-forward situation).

When two trees are specified, the user is telling *git read-tree* the following:

1. The current index and work tree is derived from $H, but the user may have local changes in them since $H.
2. The user wants to fast-forward to $M.

In this case, the `git read-tree -m $H $M` command makes sure that no local change is lost as the result of this "merge". Here are the "carry forward" rules, where "I" denotes the index, "clean" means that index and work tree coincide, and "exists"/"nothing" refer to the presence of a path in the specified commit:

```
I                   H        M        Result
   -------------------------------------------------------
 0  nothing             nothing  nothing  (does not happen)
 1  nothing             nothing  exists   use M
 2  nothing             exists   nothing  remove path from index
 3  nothing             exists   exists,  use M if "initial checkout",
                  H == M   keep index otherwise
                  exists,  fail
                  H != M
```

```
   clean I==H  I==M
  ------------------
4  yes   N/A   N/A     nothing  nothing  keep index
5  no    N/A   N/A     nothing  nothing  keep index
```

```
6  yes   N/A   yes     nothing  exists   keep index
7  no    N/A   yes     nothing  exists   keep index
8  yes   N/A   no      nothing  exists   fail
9  no    N/A   no      nothing  exists   fail
```

```
10 yes   yes   N/A     exists   nothing  remove path from index
11 no    yes   N/A     exists   nothing  fail
12 yes   no    N/A     exists   nothing  fail
13 no    no    N/A     exists   nothing  fail
```

```
clean (H==M)
   ------
 14 yes                 exists   exists   keep index
 15 no                  exists   exists   keep index
```

```
   clean I==H  I==M (H!=M)
  ------------------
16 yes   no    no      exists   exists   fail
17 no    no    no      exists   exists   fail
18 yes   no    yes     exists   exists   keep index
19 no    no    yes     exists   exists   keep index
20 yes   yes   no      exists   exists   use M
21 no    yes   no      exists   exists   fail
```

In all "keep index" cases, the index entry stays as in the original index file. If the entry is not up to date, *git read-tree* keeps the copy in the work tree intact when operating under the -u flag.

When this form of *git read-tree* returns successfully, you can see which of the "local changes" that you made were carried forward by running `git diff-index --cached $M`. Note that this does not necessarily match what `git diff-index --cached $H` would have produced before such a two tree merge. This is because of cases 18 and 19 --- if you already had the changes in $M (e.g. maybe you picked it up via e-mail in a patch form), `git diff-index --cached $H` would have told you about the change before this merge, but it would not show in `git diff-index --cached $M` output after the two-tree merge.

Case 3 is slightly tricky and needs explanation. The result from this rule logically should be to remove the path if the user staged the removal of the path and then switching to a new branch. That however will prevent the initial checkout from happening, so the rule is modified to use M (new tree) only when the content of the index is empty. Otherwise the removal of the path is kept as long as $H and $M are the same.

3-Way Merge

Each "index" entry has two bits worth of "stage" state. stage 0 is the normal one, and is the only one you'd see in any kind of normal use.

However, when you do *git read-tree* with three trees, the "stage" starts out at 1.

This means that you can do

```
$ git read-tree -m <tree1> <tree2> <tree3>
```

and you will end up with an index with all of the <tree1> entries in "stage1", all of the <tree2> entries in "stage2" and all of the <tree3> entries in "stage3". When performing a merge of another branch into the current branch, we use the common ancestor tree as <tree1>, the current branch head as <tree2>, and the other branch head as <tree3>.

Furthermore, *git read-tree* has special-case logic that says: if you see a file that matches in all respects in the following states, it "collapses" back to "stage0":

- stage 2 and 3 are the same; take one or the other (it makes no difference - the same work has been done on our branch in stage 2 and their branch in stage 3)
- stage 1 and stage 2 are the same and stage 3 is different; take stage 3 (our branch in stage 2 did not do anything since the ancestor in stage 1 while their branch in stage 3 worked on it)
- stage 1 and stage 3 are the same and stage 2 is different take stage 2 (we did something while they did nothing)

The *git write-tree* command refuses to write a nonsensical tree, and it will complain about unmerged entries if it sees a single entry that is not stage 0.

OK, this all sounds like a collection of totally nonsensical rules, but it’s actually exactly what you want in order to do a fast merge. The different stages represent the "result tree" (stage 0, aka "merged"), the original tree (stage 1, aka "orig"), and the two trees you are trying to merge (stage 2 and 3 respectively).

The order of stages 1, 2 and 3 (hence the order of three <tree-ish> command-line arguments) are significant when you start a 3-way merge with an index file that is already populated. Here is an outline of how the algorithm works:

- if a file exists in identical format in all three trees, it will automatically collapse to "merged" state by *git read-tree*.
- a file that has *any* difference what-so-ever in the three trees will stay as separate entries in the index. It’s up to "porcelain policy" to determine how to remove the non-0 stages, and insert a merged version.
- the index file saves and restores with all this information, so you can merge things incrementally, but as long as it has entries in stages 1/2/3 (i.e., "unmerged entries") you can’t write the result. So now the merge algorithm ends up being really simple:
    - you walk the index in order, and ignore all entries of stage 0, since they’ve already been done.
    - if you find a "stage1", but no matching "stage2" or "stage3", you know it’s been removed from both trees (it only existed in the original tree), and you remove that entry.
    - if you find a matching "stage2" and "stage3" tree, you remove one of them, and turn the other into a "stage0" entry. Remove any matching "stage1" entry if it exists too. .. all the normal trivial rules ..

You would normally use *git merge-index* with supplied *git merge-one-file* to do this last step. The script updates the files in the working tree as it merges each path and at the end of a successful merge.

When you start a 3-way merge with an index file that is already populated, it is assumed that it represents the state of the files in your work tree, and you can even have files with changes unrecorded in the index file. It is further assumed that this state is "derived" from the stage 2 tree. The 3-way merge refuses to run if it finds an entry in the original index file that does not match stage 2.

This is done to prevent you from losing your work-in-progress changes, and mixing your random changes in an unrelated merge commit. To illustrate, suppose you start from what has been committed last to your repository:

```
$ JC=`git rev-parse --verify "HEAD^0"`
$ git checkout-index -f -u -a $JC
```

You do random edits, without running *git update-index*. And then you notice that the tip of your "upstream" tree has advanced since you pulled from him:

```
$ git fetch git://.... linus
$ LT=`git rev-parse FETCH_HEAD`
```

Your work tree is still based on your HEAD ($JC), but you have some edits since. Three-way merge makes sure that you have not added or modified index entries since $JC, and if you haven’t, then does the right thing. So with the following sequence:

```
$ git read-tree -m -u `git merge-base $JC $LT` $JC $LT
$ git merge-index git-merge-one-file -a
$ echo "Merge with Linus" | \
  git commit-tree `git write-tree` -p $JC -p $LT
```

what you would commit is a pure merge between $JC and $LT without your work-in-progress changes, and your work tree would be updated to the result of the merge.

However, if you have local changes in the working tree that would be overwritten by this merge, *git read-tree* will refuse to run to prevent your changes from being lost.

In other words, there is no need to worry about what exists only in the working tree. When you have local changes in a part of the project that is not involved in the merge, your changes do not interfere with the merge, and are kept intact. When they **do** interfere, the merge does not even start (*git read-tree* complains loudly and fails without modifying

anything). In such a case, you can simply continue doing what you were in the middle of doing, and when your working tree is ready (i.e. you have finished your work-in-progress), attempt the merge again.

Sparse checkout

"Sparse checkout" allows populating the working directory sparsely. It uses the skip-worktree bit (see git-update-index[1]) to tell Git whether a file in the working directory is worth looking at.

git read-tree and other merge-based commands (*git merge*, *git checkout*…) can help maintaining the skip-worktree bitmap and working directory update. `$GIT_DIR/info/sparse-checkout` is used to define the skip-worktree reference bitmap. When *git read-tree* needs to update the working directory, it resets the skip-worktree bit in the index based on this file, which uses the same syntax as .gitignore files. If an entry matches a pattern in this file, skip-worktree will not be set on that entry. Otherwise, skip-worktree will be set.

Then it compares the new skip-worktree value with the previous one. If skip-worktree turns from set to unset, it will add the corresponding file back. If it turns from unset to set, that file will be removed.

While `$GIT_DIR/info/sparse-checkout` is usually used to specify what files are in, you can also specify what files are *not* in, using negate patterns. For example, to remove the file `unwanted`:

```
/*
!unwanted
```

Another tricky thing is fully repopulating the working directory when you no longer want sparse checkout. You cannot just disable "sparse checkout" because skip-worktree bits are still in the index and your working directory is still sparsely populated. You should re-populate the working directory with the `$GIT_DIR/info/sparse-checkout` file content as follows:

```
/*
```

Then you can disable sparse checkout. Sparse checkout support in *git read-tree* and similar commands is disabled by default. You need to turn `core.sparseCheckout` on in order to have sparse checkout support.

SEE ALSO

git-write-tree[1]; git-ls-files[1]; gitignore[5]

GIT

Part of the git[1] suite

rev-list

NAME

git-rev-list - Lists commit objects in reverse chronological order

SYNOPSIS

```
git rev-list [ --max-count=<number> ]
         [ --skip=<number> ]
         [ --max-age=<timestamp> ]
         [ --min-age=<timestamp> ]
         [ --sparse ]
         [ --merges ]
         [ --no-merges ]
         [ --min-parents=<number> ]
         [ --no-min-parents ]
         [ --max-parents=<number> ]
         [ --no-max-parents ]
         [ --first-parent ]
         [ --remove-empty ]
         [ --full-history ]
         [ --not ]
         [ --all ]
         [ --branches[=<pattern>] ]
         [ --tags[=<pattern>] ]
         [ --remotes[=<pattern>] ]
         [ --glob=<glob-pattern> ]
         [ --ignore-missing ]
         [ --stdin ]
         [ --quiet ]
         [ --topo-order ]
         [ --parents ]
         [ --timestamp ]
         [ --left-right ]
         [ --left-only ]
         [ --right-only ]
         [ --cherry-mark ]
         [ --cherry-pick ]
         [ --encoding=<encoding> ]
         [ --(author|committer|grep)=<pattern> ]
         [ --regexp-ignore-case | -i ]
         [ --extended-regexp | -E ]
         [ --fixed-strings | -F ]
         [ --date=<format>]
         [ [ --objects | --objects-edge | --objects-edge-aggressive ]
           [ --unpacked ] ]
         [ --pretty | --header ]
         [ --bisect ]
         [ --bisect-vars ]
         [ --bisect-all ]
         [ --merge ]
         [ --reverse ]
         [ --walk-reflogs ]
         [ --no-walk ] [ --do-walk ]
         [ --count ]
         [ --use-bitmap-index ]
         <commit>… [ -- <paths>… ]
```

DESCRIPTION

List commits that are reachable by following the `parent` links from the given commit(s), but exclude commits that are reachable from the one(s) given with a ^ in front of them. The output is given in reverse chronological order by default.

You can think of this as a set operation. Commits given on the command line form a set of commits that are reachable from any of them, and then commits reachable from any of the ones given with ^ in front are subtracted from that set. The remaining commits are what comes out in the command's output. Various other options and paths parameters can be used to further limit the result.

Thus, the following command:

```
$ git rev-list foo bar ^baz
```

means "list all the commits which are reachable from *foo* or *bar*, but not from *baz*".

A special notation "*<commit1>*..*<commit2>*" can be used as a short-hand for "^'<commit1>' *<commit2>*". For example, either of the following may be used interchangeably:

```
$ git rev-list origin..HEAD
$ git rev-list HEAD ^origin
```

Another special notation is "*<commit1>*…*<commit2>*" which is useful for merges. The resulting set of commits is the symmetric difference between the two operands. The following two commands are equivalent:

```
$ git rev-list A B --not $(git merge-base --all A B)
$ git rev-list A...B
```

rev-list is a very essential Git command, since it provides the ability to build and traverse commit ancestry graphs. For this reason, it has a lot of different options that enables it to be used by commands as different as *git bisect* and *git repack*.

OPTIONS

Commit Limiting

Besides specifying a range of commits that should be listed using the special notations explained in the description, additional commit limiting may be applied.

Using more options generally further limits the output (e.g. `--since=<date1>` limits to commits newer than `<date1>`, and using it with `--grep=<pattern>` further limits to commits whose log message has a line that matches `<pattern>`), unless otherwise noted.

Note that these are applied before commit ordering and formatting options, such as `--reverse`.

-<number>

-n <number>

--max-count=<number>

Limit the number of commits to output.

--skip=<number>

Skip *number* commits before starting to show the commit output.

--since=<date>

--after=<date>

Show commits more recent than a specific date.

--until=<date>

--before=<date>

Show commits older than a specific date.

--max-age=<timestamp>

--min-age=<timestamp>

Limit the commits output to specified time range.

--author=<pattern>

--committer=<pattern>

Limit the commits output to ones with author/committer header lines that match the specified pattern (regular expression). With more than one `--author=<pattern>`, commits whose author matches any of the given patterns are chosen (similarly for multiple `--committer=<pattern>`).

--grep-reflog=<pattern>

Limit the commits output to ones with reflog entries that match the specified pattern (regular expression). With more than one `--grep-reflog`, commits whose reflog message matches any of the given patterns are chosen. It is an error to use this option unless `--walk-reflogs` is in use.

--grep=<pattern>

Limit the commits output to ones with log message that matches the specified pattern (regular expression). With more than one `--grep=<pattern>`, commits whose message matches any of the given patterns are chosen (but see `--all-match`).

--all-match

Limit the commits output to ones that match all given `--grep`, instead of ones that match at least one.

--invert-grep

Limit the commits output to ones with log message that do not match the pattern specified with `--grep=<pattern>`.

-i

--regexp-ignore-case

Match the regular expression limiting patterns without regard to letter case.

--basic-regexp

Consider the limiting patterns to be basic regular expressions; this is the default.

-E

--extended-regexp

Consider the limiting patterns to be extended regular expressions instead of the default basic regular expressions.

-F

--fixed-strings

Consider the limiting patterns to be fixed strings (don’t interpret pattern as a regular expression).

--perl-regexp

Consider the limiting patterns to be Perl-compatible regular expressions. Requires libpcre to be compiled in.

--remove-empty

Stop when a given path disappears from the tree.

--merges

Print only merge commits. This is exactly the same as `--min-parents=2`.

--no-merges

Do not print commits with more than one parent. This is exactly the same as `--max-parents=1`.

--min-parents=<number>

--max-parents=<number>

--no-min-parents

--no-max-parents

Show only commits which have at least (or at most) that many parent commits. In particular, `--max-parents=1` is the same as `--no-merges`, `--min-parents=2` is the same as `--merges`. `--max-parents=0` gives all root commits and `--min-parents=3` all octopus merges.

`--no-min-parents` and `--no-max-parents` reset these limits (to no limit) again. Equivalent forms are `--min-parents=0` (any commit has 0 or more parents) and `--max-parents=-1` (negative numbers denote no upper limit).

--first-parent

Follow only the first parent commit upon seeing a merge commit. This option can give a better overview when viewing the evolution of a particular topic branch, because merges into a topic branch tend to be only about adjusting to updated upstream from time to time, and this option allows you to ignore the individual commits brought in to your history by such a merge. Cannot be combined with --bisect.

--not

Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next `--not`.

--all

Pretend as if all the refs in `refs/` are listed on the command line as <*commit*>.

--branches[=<pattern>]

Pretend as if all the refs in `refs/heads` are listed on the command line as *<commit>*. If *<pattern>* is given, limit branches to ones matching given shell glob. If pattern lacks *?*, *, or *[*, */** at the end is implied.

--tags[=<pattern>]

Pretend as if all the refs in `refs/tags` are listed on the command line as *<commit>*. If *<pattern>* is given, limit tags to ones matching given shell glob. If pattern lacks *?*, *, or *[*, */** at the end is implied.

--remotes[=<pattern>]

Pretend as if all the refs in `refs/remotes` are listed on the command line as *<commit>*. If *<pattern>* is given, limit remote-tracking branches to ones matching given shell glob. If pattern lacks *?*, *, or *[*, */** at the end is implied.

--glob=<glob-pattern>

Pretend as if all the refs matching shell glob *<glob-pattern>* are listed on the command line as *<commit>*. Leading *refs/*, is automatically prepended if missing. If pattern lacks *?*, *, or *[*, */** at the end is implied.

--exclude=<glob-pattern>

Do not include refs matching *<glob-pattern>* that the next `--all`, `--branches`, `--tags`, `--remotes`, or `--glob` would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next `--all`, `--branches`, `--tags`, `--remotes`, or `--glob` option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with `refs/heads`, `refs/tags`, or `refs/remotes` when applied to `--branches`, `--tags`, or `--remotes`, respectively, and they must begin with `refs/` when applied to `--glob` or `--all`. If a trailing */** is intended, it must be given explicitly.

--reflog

Pretend as if all objects mentioned by reflogs are listed on the command line as `<commit>`.

--ignore-missing

Upon seeing an invalid object name in the input, pretend as if the bad input was not given.

--stdin

In addition to the *<commit>* listed on the command line, read them from the standard input. If a -- separator is seen, stop reading commits and start reading paths to limit the result.

--quiet

Don’t print anything to standard output. This form is primarily meant to allow the caller to test the exit status to see if a range of objects is fully connected (or not). It is faster than redirecting stdout to `/dev/null` as the output does not have to be formatted.

--cherry-mark

Like `--cherry-pick` (see below) but mark equivalent commits with `=` rather than omitting them, and inequivalent ones with `+`.

--cherry-pick

Omit any commit that introduces the same change as another commit on the “other side” when the set of commits are limited with symmetric difference.

For example, if you have two branches, `A` and `B`, a usual way to list all commits on only one side of them is with `--left-right` (see the example below in the description of the `--left-right` option). However, it shows the commits that were cherry-picked from the other branch (for example, “3rd on b” may be cherry-picked from branch A). With this option, such pairs of commits are excluded from the output.

--left-only

--right-only

List only commits on the respective side of a symmetric range, i.e. only those which would be marked `<` resp. `>` by `--left-right`.

For example, `--cherry-pick --right-only A...B` omits those commits from `B` which are in `A` or are patch-equivalent to a commit in `A`. In other words, this lists the `+` commits from `git cherry A B`. More precisely, `--cherry-pick --right-only --no-merges` gives the exact list.

--cherry

A synonym for `--right-only --cherry-mark --no-merges`; useful to limit the output to the commits on our side and mark those that have been applied to the other side of a forked history with `git log --cherry upstream...mybranch`, similar to `git cherry upstream mybranch`.

-g

--walk-reflogs

Instead of walking the commit ancestry chain, walk reflog entries from the most recent one to older ones. When this option is used you cannot specify commits to exclude (that is, *^commit*, *commit1..commit2*, and *commit1...commit2* notations cannot be used).

With `--pretty` format other than `oneline` (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. By default, *commit@{Nth}* notation is used in the output. When the starting commit is specified as *commit@{now}*, output also uses *commit@{timestamp}* notation instead. Under `--pretty=oneline` , the commit message is prefixed with this information on the same line. This option cannot be combined with `--reverse` . See also git-reflog[1].

--merge

After a failed merge, show refs that touch files having a conflict and don't exist on all heads to merge.

--boundary

Output excluded boundary commits. Boundary commits are prefixed with `-` .

--use-bitmap-index

Try to speed up the traversal using the pack bitmap index (if one is available). Note that when traversing with `--objects` , trees and blobs will not have their associated path printed.

History Simplification

Sometimes you are only interested in parts of the history, for example the commits modifying a particular <path>. But there are two parts of *History Simplification*, one part is selecting the commits and the other is how to do it, as there are various strategies to simplify the history.

The following options select the commits to be shown:

<paths>

Commits modifying the given <paths> are selected.

--simplify-by-decoration

Commits that are referred by some branch or tag are selected.

Note that extra commits can be shown to give a meaningful history.

The following options affect the way the simplification is performed:

Default mode

Simplifies the history to the simplest history explaining the final state of the tree. Simplest because it prunes some side branches if the end result is the same (i.e. merging branches with the same content)

--full-history

Same as the default mode, but does not prune some history.

--dense

Only the selected commits are shown, plus some to have a meaningful history.

--sparse

All commits in the simplified history are shown.

--simplify-merges

Additional option to `--full-history` to remove some needless merges from the resulting history, as there are no selected commits contributing to this merge.

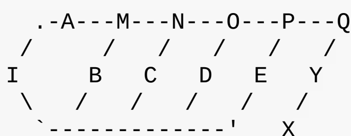
--ancestry-path

When given a range of commits to display (e.g. *commit1..commit2* or *commit2 ^commit1*), only display commits that exist directly on the ancestry chain between the *commit1* and *commit2*, i.e. commits that are both descendants of *commit1*, and ancestors of *commit2*.

A more detailed explanation follows.

Suppose you specified `foo` as the <paths>. We shall call commits that modify `foo` !TREESAME, and the rest TREESAME. (In a diff filtered for `foo`, they look different and equal, respectively.)

In the following, we will always refer to the same example history to illustrate the differences between simplification settings. We assume that you are filtering for a file `foo` in this commit graph:



```
  .-A---M---N---O---P---Q
 /     /   /   /   /   /
I     B   C   D   E   Y
 \   /   /   /   /   /
  `-------------'   X
```

The horizontal line of history A---Q is taken to be the first parent of each merge. The commits are:

- `I` is the initial commit, in which `foo` exists with contents “asdf”, and a file `quux` exists with contents “quux”. Initial commits are compared to an empty tree, so `I` is !TREESAME.
- In `A`, `foo` contains just “foo”.
- `B` contains the same change as `A`. Its merge `M` is trivial and hence TREESAME to all parents.

- `C` does not change `foo`, but its merge `N` changes it to “foobar”, so it is not TREESAME to any parent.
- `D` sets `foo` to “baz”. Its merge `O` combines the strings from `N` and `D` to “foobarbaz”; i.e., it is not TREESAME to any parent.
- `E` changes `quux` to “xyzzy”, and its merge `P` combines the strings to “quux xyzzy”. `P` is TREESAME to `O`, but not to `E`.
- `X` is an independent root commit that added a new file `side`, and `Y` modified it. `Y` is TREESAME to `X`. Its merge `Q` added `side` to `P`, and `Q` is TREESAME to `P`, but not to `Y`.

`rev-list` walks backwards through history, including or excluding commits based on whether `--full-history` and/or parent rewriting (via `--parents` or `--children`) are used. The following settings are available.

Default mode

Commits are included if they are not TREESAME to any parent (though this can be changed, see `--sparse` below). If the commit was a merge, and it was TREESAME to one parent, follow only that parent. (Even if there are several TREESAME parents, follow only one of them.) Otherwise, follow all parents.

This results in:

```
  .-A---N---O
 /     /   /
I---------D
```

Note how the rule to only follow the TREESAME parent, if one is available, removed `B` from consideration entirely. `C` was considered via `N`, but is TREESAME. Root commits are compared to an empty tree, so `I` is !TREESAME.

Parent/child relations are only visible with `--parents`, but that does not affect the commits selected in default mode, so we have shown the parent lines.

--full-history without parent rewriting

This mode differs from the default in one point: always follow all parents of a merge, even if it is TREESAME to one of them. Even if more than one side of the merge has commits that are included, this does not imply that the merge itself is! In the example, we get

```
I  A  B  N  D  O  P  Q
```

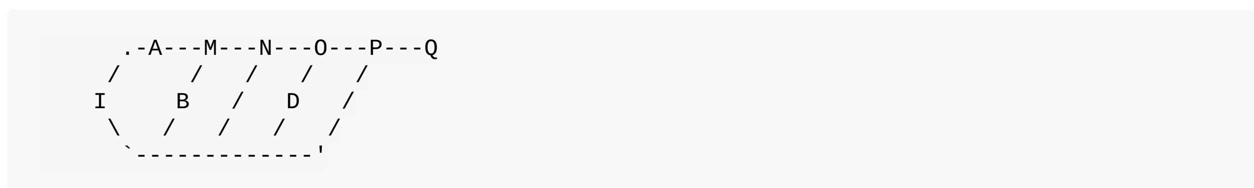
`M` was excluded because it is TREESAME to both parents. `E`, `C` and `B` were all walked, but only `B` was !TREESAME, so the others do not appear.

Note that without parent rewriting, it is not really possible to talk about the parent/child relationships between the commits, so we show them disconnected.

--full-history with parent rewriting

Ordinary commits are only included if they are !TREESAME (though this can be changed, see `--sparse` below).

Merges are always included. However, their parent list is rewritten: Along each parent, prune away commits that are not included themselves. This results in



```
	  .-A---M---N---O---P---Q
	 /     /   /   /   /
	I     B   /   D   /
	 \   /   /   /   /
	  `-------------'
```

Compare to `--full-history` without rewriting above. Note that `E` was pruned away because it is TREESAME, but the parent list of P was rewritten to contain `E`'s parent `I`. The same happened for `C` and `N`, and `X`, `Y` and `Q`.

In addition to the above settings, you can change whether TREESAME affects inclusion:

--dense

Commits that are walked are included if they are not TREESAME to any parent.

--sparse

All commits that are walked are included.

Note that without `--full-history`, this still simplifies merges: if one of the parents is TREESAME, we follow only that one, so the other sides of the merge are never walked.

--simplify-merges

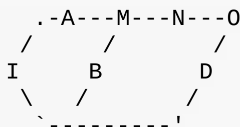
First, build a history graph in the same way that `--full-history` with parent rewriting does (see above).

Then simplify each commit `C` to its replacement `C'` in the final history according to the following rules:

- Set `C'` to `C`.

- Replace each parent `P` of `C'` with its simplification `P'`. In the process, drop parents that are ancestors of other parents or that are root commits TREESAME to an empty tree, and remove duplicates, but take care to never drop all parents that we are TREESAME to.
- If after this parent rewriting, `C'` is a root or merge commit (has zero or >1 parents), a boundary commit, or !TREESAME, it remains. Otherwise, it is replaced with its only parent.

The effect of this is best shown by way of comparing to `--full-history` with parent rewriting. The example turns into:



Note the major differences in `N`, `P`, and `Q` over `--full-history`:

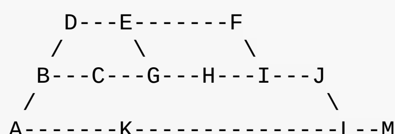
- `N`'s parent list had `I` removed, because it is an ancestor of the other parent `M`. Still, `N` remained because it is !TREESAME.
- `P`'s parent list similarly had `I` removed. `P` was then removed completely, because it had one parent and is TREESAME.
- `Q`'s parent list had `Y` simplified to `X`. `X` was then removed, because it was a TREESAME root. `Q` was then removed completely, because it had one parent and is TREESAME.

Finally, there is a fifth simplification mode available:

--ancestry-path

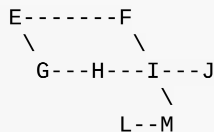
Limit the displayed commits to those directly on the ancestry chain between the “from” and “to” commits in the given commit range. I.e. only display commits that are ancestor of the “to” commit and descendants of the “from” commit.

As an example use case, consider the following commit history:



A regular *D..M* computes the set of commits that are ancestors of `M`, but excludes the ones that are ancestors of `D`. This is useful to see what happened to the history leading to `M` since `D`, in the sense that “what does `M` have that did not exist in `D`”. The result in this example would be all the commits, except `A` and `B` (and `D` itself, of course).

When we want to find out what commits in `M` are contaminated with the bug introduced by `D` and need fixing, however, we might want to view only the subset of *D..M* that are actually descendants of `D`, i.e. excluding `C` and `K`. This is exactly what the `--ancestry-path` option does. Applied to the *D..M* range, it results in:



```
        E-------F
         \       \
          G---H---I---J
                   \
                L--M
```

The `--simplify-by-decoration` option allows you to view only the big picture of the topology of the history, by omitting commits that are not referenced by tags. Commits are marked as !TREESAME (in other words, kept after history simplification rules described above) if (1) they are referenced by tags, or (2) they change the contents of the paths given on the command line. All other commits are marked as TREESAME (subject to be simplified away).

Bisection Helpers

--bisect

Limit output to the one commit object which is roughly halfway between included and excluded commits. Note that the bad bisection ref `refs/bisect/bad` is added to the included commits (if it exists) and the good bisection refs `refs/bisect/good-*` are added to the excluded commits (if they exist). Thus, supposing there are no refs in `refs/bisect/`, if

```
    $ git rev-list --bisect foo ^bar ^baz
```

outputs *midpoint*, the output of the two commands

```
    $ git rev-list foo ^midpoint
    $ git rev-list midpoint ^bar ^baz
```

would be of roughly the same length. Finding the change which introduces a regression is thus reduced to a binary search: repeatedly generate and test new 'midpoint’s until the commit chain is of length one. Cannot be combined with --first-parent.

--bisect-vars

This calculates the same as `--bisect`, except that refs in `refs/bisect/` are not used, and except that this outputs text ready to be eval'ed by the shell. These lines will assign the name of the midpoint revision to the variable `bisect_rev`, and the expected number of commits to be tested after `bisect_rev` is tested to `bisect_nr`, the expected number of commits to be tested if `bisect_rev` turns out to be good to `bisect_good`, the expected number of commits to be tested if `bisect_rev` turns out to be bad to `bisect_bad`, and the number of commits we are bisecting right now to `bisect_all`.

--bisect-all

This outputs all the commit objects between the included and excluded commits, ordered by their distance to the included and excluded commits. Refs in `refs/bisect/` are not used. The farthest from them is displayed first. (This is the only one displayed by `--bisect`.)

This is useful because it makes it easy to choose a good commit to test when you want to avoid to test some of them for some reason (they may not compile for example).

This option can be used along with `--bisect-vars`, in this case, after all the sorted commit objects, there will be the same text as if `--bisect-vars` had been used alone.

Commit Ordering

By default, the commits are shown in reverse chronological order.

--date-order

Show no parents before all of its children are shown, but otherwise show commits in the commit timestamp order.

--author-date-order

Show no parents before all of its children are shown, but otherwise show commits in the author timestamp order.

--topo-order

Show no parents before all of its children are shown, and avoid showing commits on multiple lines of history intermixed.

For example, in a commit history like this:

```
    ---1----2----4----7
    \           \
     3----5----6----8---
```

where the numbers denote the order of commit timestamps, `git rev-list` and friends with `--date-order` show the commits in the timestamp order: 8 7 6 5 4 3 2 1.

With `--topo-order`, they would show 8 6 5 3 7 4 2 1 (or 8 7 4 2 6 5 3 1); some older commits are shown before newer ones in order to avoid showing the commits from two parallel development track mixed together.

--reverse

Output the commits in reverse order. Cannot be combined with `--walk-reflogs`.

Object Traversal

These options are mostly targeted for packing of Git repositories.

--objects

Print the object IDs of any object referenced by the listed commits. `--objects foo ^bar` thus means "send me all object IDs which I need to download if I have the commit object *bar* but not *foo*".

--objects-edge

Similar to `--objects`, but also print the IDs of excluded commits prefixed with a "-" character. This is used by git-pack-objects[1] to build a "thin" pack, which records objects in deltified form based on objects contained in these excluded commits to reduce network traffic.

--objects-edge-aggressive

Similar to `--objects-edge`, but it tries harder to find excluded commits at the cost of increased time. This is used instead of `--objects-edge` to build "thin" packs for shallow repositories.

--indexed-objects

Pretend as if all trees and blobs used by the index are listed on the command line. Note that you probably want to use `--objects`, too.

--unpacked

Only useful with `--objects`; print the object IDs that are not in packs.

--no-walk[=(sorted|unsorted)]

Only show the given commits, but do not traverse their ancestors. This has no effect if a range is specified. If the argument `unsorted` is given, the commits are shown in the order they were given on the command line. Otherwise (if `sorted` or no argument was given), the commits are shown in reverse chronological order by commit time. Cannot be combined with `--graph`.

--do-walk

Overrides a previous `--no-walk`.

Commit Formatting

Using these options, git-rev-list[1] will act similar to the more specialized family of commit log tools: git-log[1], git-show[1], and git-whatchanged[1]

pretty-options.txt

--relative-date

Synonym for `--date=relative`.

--date=<format>

Only takes effect for dates shown in human-readable format, such as when using `--pretty`. `log.date` config variable sets a default value for the log command's `--date` option. By default, dates are shown in the original time zone (either committer's or author's). If `-local` is appended to the format (e.g., `iso-local`), the user's local time zone is used instead.

`--date=relative` shows dates relative to the current time, e.g. “2 hours ago”. The `-local` option cannot be used with `--raw` or `--relative`.

`--date=local` is an alias for `--date=default-local`.

`--date=iso` (or `--date=iso8601`) shows timestamps in a ISO 8601-like format. The differences to the strict ISO 8601 format are:

- a space instead of the `T` date/time delimiter
- a space between time and time zone
- no colon between hours and minutes of the time zone

`--date=iso-strict` (or `--date=iso8601-strict`) shows timestamps in strict ISO 8601 format.

- `--date=rfc` (or `--date=rfc2822`) shows timestamps in RFC 2822 format, often found in email messages.
- `--date=short` shows only the date, but not the time, in `YYYY-MM-DD` format.
- `--date=raw` shows the date in the internal raw Git format `%s %z` format.
- `--date=format:...` feeds the format `...` to your system `strftime`. Use `--date=format:%c` to show the date in your system locale's preferred format. See the `strftime` manual for a complete list of format placeholders. When using `-local`, the correct syntax is `--date=format-local:...`.

- `--date=default` is the default format, and is similar to `--date=rfc2822`, with a few exceptions:
- there is no comma after the day-of-week
- the time zone is omitted when the local time zone is used

--header

Print the contents of the commit in raw-format; each record is separated with a NUL character.

--parents

Print also the parents of the commit (in the form "commit parent…"). Also enables parent rewriting, see *History Simplification* below.

--children

Print also the children of the commit (in the form "commit child…"). Also enables parent rewriting, see *History Simplification* below.

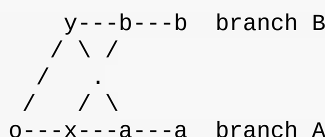
--timestamp

Print the raw commit timestamp.

--left-right

Mark which side of a symmetric diff a commit is reachable from. Commits from the left side are prefixed with `<` and those from the right with `>`. If combined with `--boundary`, those commits are prefixed with `-`.

For example, if you have this topology:



```
             y---b---b  branch B
            / \ /
           /   .
          /   / \
         o---x---a---a  branch A
```

you would get an output like this:

```
$ git rev-list --left-right --boundary --pretty=oneline A...B

&gt;bbbbbbb... 3rd on b
&gt;bbbbbbb... 2nd on b
&lt;aaaaaaa... 3rd on a
&lt;aaaaaaa... 2nd on a
-yyyyyyy... 1st on b
-xxxxxxx... 1st on a
```

--graph

Draw a text-based graphical representation of the commit history on the left hand side of the output. This may cause extra lines to be printed in between commits, in order for the graph history to be drawn properly. Cannot be combined with `--no-walk`.

This enables parent rewriting, see *History Simplification* below.

This implies the `--topo-order` option by default, but the `--date-order` option may also be specified.

--show-linear-break[=<barrier>]

When --graph is not used, all history branches are flattened which can make it hard to see that the two consecutive commits do not belong to a linear branch. This option puts a barrier in between them in that case. If `<barrier>` is specified, it is the string that will be shown instead of the default one.

--count

Print a number stating how many commits would have been listed, and suppress all other output. When used together with `--left-right`, instead print the counts for left and right commits, separated by a tab. When used together with `--cherry-mark`, omit patch equivalent commits from these counts and print the count for equivalent commits separated by a tab.

PRETTY FORMATS

If the commit is a merge, and if the pretty-format is not *oneline*, *email* or *raw*, an additional line is inserted before the *Author:* line. This line begins with "Merge: " and the sha1s of ancestral commits are printed, separated by spaces. Note that the listed commits may not necessarily be the list of the **direct** parent commits if you have limited your view of history: for example, if you are only interested in changes related to a certain directory or file.

There are several built-in formats, and you can define additional formats by setting a pretty.<name> config option to either another format name, or a *format:* string, as described below (see git-config[1]). Here are the details of the built-in formats:

- *oneline*

  ```
  &lt;sha1&gt; &lt;title line&gt;
  ```

  This is designed to be as compact as possible.

- *short*

```
commit &lt;sha1&gt;
Author: &lt;author&gt;
```

```
&lt;title line&gt;
```

- *medium*

  ```
  commit &lt;sha1&gt;
  Author: &lt;author&gt;
  Date:   &lt;author date&gt;
  ```

  ```
  &lt;title line&gt;
  ```

  ```
  &lt;full commit message&gt;
  ```

- *full*

  ```
  commit &lt;sha1&gt;
  Author: &lt;author&gt;
  Commit: &lt;committer&gt;
  ```

  ```
  &lt;title line&gt;
  ```

  ```
  &lt;full commit message&gt;
  ```

- *fuller*

  ```
  commit &lt;sha1&gt;
  Author:     &lt;author&gt;
  AuthorDate: &lt;author date&gt;
  Commit:     &lt;committer&gt;
  CommitDate: &lt;committer date&gt;
  ```

  ```
  &lt;title line&gt;
  ```

  ```
  &lt;full commit message&gt;
  ```

- *email*

```
From &lt;sha1&gt; &lt;date&gt;
From: &lt;author&gt;
Date: &lt;author date&gt;
Subject: [PATCH] &lt;title line&gt;
```

```
&lt;full commit message&gt;
```

- *raw*

  The *raw* format shows the entire commit exactly as stored in the commit object. Notably, the SHA-1s are displayed in full, regardless of whether --abbrev or --no-abbrev are used, and *parents* information show the true parent commits, without taking grafts or history simplification into account. Note that this format affects the way commits are displayed, but not the way the diff is shown e.g. with `git log --raw`. To get full object names in a raw diff format, use `--no-abbrev`.

- *format:<string>*

  The *format:<string>* format allows you to specify which information you want to show. It works a little bit like printf format, with the notable exception that you get a newline with *%n* instead of *\n*.

  E.g, *format:"The author of %h was %an, %ar%nThe title was >>%s<<%n"* would show something like this:

  ```
  The author of fe6e0ee was Junio C Hamano, 23 hours ago
  The title was &gt;&gt;t4119: test autocomputing -p&lt;n&gt; for traditional diff inpu
  ```

  The placeholders are:

  - *%H*: commit hash
  - *%h*: abbreviated commit hash
  - *%T*: tree hash
  - *%t*: abbreviated tree hash
  - *%P*: parent hashes
  - *%p*: abbreviated parent hashes
  - *%an*: author name
  - *%aN*: author name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
  - *%ae*: author email

- *%aE*: author email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ad*: author date (format respects --date= option)
- *%aD*: author date, RFC2822 style
- *%ar*: author date, relative
- *%at*: author date, UNIX timestamp
- *%ai*: author date, ISO 8601-like format
- *%aI*: author date, strict ISO 8601 format
- *%cn*: committer name
- *%cN*: committer name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ce*: committer email
- *%cE*: committer email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%cd*: committer date (format respects --date= option)
- *%cD*: committer date, RFC2822 style
- *%cr*: committer date, relative
- *%ct*: committer date, UNIX timestamp
- *%ci*: committer date, ISO 8601-like format
- *%cI*: committer date, strict ISO 8601 format
- *%d*: ref names, like the --decorate option of git-log[1]
- *%D*: ref names without the " (", ")" wrapping.
- *%e*: encoding
- *%s*: subject
- *%f*: sanitized subject line, suitable for a filename
- *%b*: body
- *%B*: raw body (unwrapped subject and body)
- *%GG*: raw verification message from GPG for a signed commit
- *%G?*: show "G" for a Good signature, "B" for a Bad signature, "U" for a good, untrusted signature and "N" for no signature

- *%GS*: show the name of the signer for a signed commit
- *%GK*: show the key used to sign a signed commit
- *%gD*: reflog selector, e.g., `refs/stash@{1}`
- *%gd*: shortened reflog selector, e.g., `stash@{1}`
- *%gn*: reflog identity name
- *%gN*: reflog identity name (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%ge*: reflog identity email
- *%gE*: reflog identity email (respecting .mailmap, see git-shortlog[1] or git-blame[1])
- *%gs*: reflog subject
- *%Cred*: switch color to red
- *%Cgreen*: switch color to green
- *%Cblue*: switch color to blue
- *%Creset*: reset color
- *%C(…)*: color specification, as described in color.branch.* config option; adding `auto,` at the beginning will emit color only when colors are enabled for log output (by `color.diff`, `color.ui`, or `--color`, and respecting the `auto` settings of the former if we are going to a terminal). `auto` alone (i.e. `%C(auto)`) will turn on auto coloring on the next placeholders until the color is switched again.
- *%m*: left, right or boundary mark
- *%n*: newline
- *%%*: a raw %
- *%x00*: print a byte from a hex code
- *%w([<w>[,<i1>[,<i2>]]])*: switch line wrapping, like the -w option of git-shortlog[1].
- *%<(<N>[,trunc|ltrunc|mtrunc])*: make the next placeholder take at least N columns, padding spaces on the right if necessary. Optionally truncate at the beginning (ltrunc), the middle (mtrunc) or the end (trunc) if the output is longer than N columns. Note that truncating only works correctly with N >= 2.
- *%<|(<N>)*: make the next placeholder take at least until Nth columns, padding spaces on the right if necessary

- *%>(<N>)*, *%>|(<N>)*: similar to *%<(<N>)*, *%<|(<N>)* respectively, but padding spaces on the left
- *%>>(<N>)*, *%>>|(<N>)*: similar to *%>(<N>)*, *%>|(<N>)* respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces
- *%><(<N>)*, *%><|(<N>)*: similar to *% <(<N>)*, *%<|(<N>)* respectively, but padding both sides (i.e. the text is centered)

> Note
>
> Some placeholders may depend on other options given to the revision traversal engine. For example, the `%g*` reflog options will insert an empty string unless we are traversing reflog entries (e.g., by `git log -g`). The `%d` and `%D` placeholders will use the "short" decoration format if `--decorate` was not already provided on the command line.

If you add a `+` (plus sign) after *%* of a placeholder, a line-feed is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

If you add a `-` (minus sign) after *%* of a placeholder, line-feeds that immediately precede the expansion are deleted if and only if the placeholder expands to an empty string.

If you add a ` ` (space) after *%* of a placeholder, a space is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

- *tformat:*

  The *tformat:* format works exactly like *format:*, except that it provides "terminator" semantics instead of "separator" semantics. In other words, each commit has the message terminator character (usually a newline) appended, rather than a separator placed between entries. This means that the final entry of a single-line format will be properly terminated with a new line, just as the "oneline" format does. For example:

  ```
  $ git log -2 --pretty=format:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973 -- NO NEWLINE

  $ git log -2 --pretty=tformat:%h 4da45bef \
    | perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'
  4da45be
  7134973
  ```

  In addition, any unrecognized string that has a `%` in it is interpreted as if it has `tformat:` in front of it. For example, these two are equivalent:

```
$ git log -2 --pretty=tformat:%h 4da45bef
$ git log -2 --pretty=%h 4da45bef
```

GIT

Part of the git[1] suite

rev-parse

NAME

git-rev-parse - Pick out and massage parameters

SYNOPSIS

```
git rev-parse [ --option ] <args>…
```

DESCRIPTION

Many Git porcelainish commands take mixture of flags (i.e. parameters that begin with a dash -) and parameters meant for the underlying *git rev-list* command they use internally and flags and parameters for the other commands they use downstream of *git rev-list*. This command is used to distinguish between them.

OPTIONS

Operation Modes

Each of these options must appear first on the command line.

--parseopt

Use *git rev-parse* in option parsing mode (see PARSEOPT section below).

--sq-quote

Use *git rev-parse* in shell quoting mode (see SQ-QUOTE section below). In contrast to the `--sq` option below, this mode does only quoting. Nothing else is done to command input.

Options for --parseopt

--keep-dashdash

Only meaningful in `--parseopt` mode. Tells the option parser to echo out the first `--` met instead of skipping it.

--stop-at-non-option

Only meaningful in `--parseopt` mode. Lets the option parser stop at the first non-option argument. This can be used to parse sub-commands that take options themselves.

--stuck-long

Only meaningful in `--parseopt` mode. Output the options in their long form if available, and with their arguments stuck.

Options for Filtering

--revs-only

Do not output flags and parameters not meant for *git rev-list* command.

--no-revs

Do not output flags and parameters meant for *git rev-list* command.

--flags

Do not output non-flag parameters.

--no-flags

Do not output flag parameters.

Options for Output

--default <arg>

If there is no parameter given by the user, use `<arg>` instead.

--prefix <arg>

Behave as if *git rev-parse* was invoked from the `<arg>` subdirectory of the working tree. Any relative filenames are resolved as if they are prefixed by `<arg>` and will be printed in that form.

This can be used to convert arguments to a command run in a subdirectory so that they can still be used after moving to the top-level of the repository. For example:

```
prefix=$(git rev-parse --show-prefix)
cd "$(git rev-parse --show-toplevel)"
eval "set -- $(git rev-parse --sq --prefix "$prefix" "$@")"
```

--verify

Verify that exactly one parameter is provided, and that it can be turned into a raw 20-byte SHA-1 that can be used to access the object database. If so, emit it to the standard output; otherwise, error out.

If you want to make sure that the output actually names an object in your object database and/or can be used as a specific type of object you require, you can add the `^{type}` peeling operator to the parameter. For example, `git rev-parse "$VAR^{commit}"` will make sure `$VAR` names an existing object that is a commit-ish (i.e. a commit, or an annotated tag that points at a commit). To make sure that `$VAR` names an existing object of any type, `git rev-parse "$VAR^{object}"` can be used.

-q

--quiet

Only meaningful in `--verify` mode. Do not output an error message if the first argument is not a valid object name; instead exit with non-zero status silently. SHA-1s for valid object names are printed to stdout on success.

--sq

Usually the output is made one line per flag and parameter. This option makes output a single line, properly quoted for consumption by shell. Useful when you expect your parameter to contain whitespaces and newlines (e.g. when using pickaxe `-S` with *git diff-**). In contrast to the `--sq-quote` option, the command input is still interpreted as usual.

--not

When showing object names, prefix them with ^ and strip ^ prefix from the object names that already have one.

--abbrev-ref[=(strict|loose)]

A non-ambiguous short name of the objects name. The option core.warnAmbiguousRefs is used to select the strict abbreviation mode.

--short

--short=number

Instead of outputting the full SHA-1 values of object names try to abbreviate them to a shorter unique name. When no length is specified 7 is used. The minimum length is 4.

--symbolic

Usually the object names are output in SHA-1 form (with possible ^ prefix); this option makes them output in a form as close to the original input as possible.

--symbolic-full-name

This is similar to --symbolic, but it omits input that are not refs (i.e. branch or tag names; or more explicitly disambiguating "heads/master" form, when you want to name the "master" branch when there is an unfortunately named tag "master"), and show them as full refnames (e.g. "refs/heads/master").

Options for Objects

--all

Show all refs found in `refs/`.

--branches[=pattern]

--tags[=pattern]

--remotes[=pattern]

Show all branches, tags, or remote-tracking branches, respectively (i.e., refs found in `refs/heads`, `refs/tags`, or `refs/remotes`, respectively).

If a `pattern` is given, only refs matching the given shell glob are shown. If the pattern does not contain a globbing character (`?`, `*`, or `[`), it is turned into a prefix match by appending `/*`.

--glob=pattern

Show all refs matching the shell glob pattern `pattern`. If the pattern does not start with `refs/`, this is automatically prepended. If the pattern does not contain a globbing character (`?`, `*`, or `[`), it is turned into a prefix match by appending `/*`.

--exclude=<glob-pattern>

Do not include refs matching *<glob-pattern>* that the next `--all`, `--branches`, `--tags`, `--remotes`, or `--glob` would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next `--all`, `--branches`, `--tags`, `--remotes`, or `--glob` option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with `refs/heads`, `refs/tags`, or `refs/remotes` when applied to `--branches`, `--tags`, or `--remotes`, respectively, and they must begin with `refs/` when applied to `--glob` or `--all`. If a trailing /* is intended, it must be given explicitly.

--disambiguate=<prefix>

Show every object whose name begins with the given prefix. The <prefix> must be at least 4 hexadecimal digits long to avoid listing each and every object in the repository by mistake.

Options for Files

--local-env-vars

List the GIT_* environment variables that are local to the repository (e.g. GIT_DIR or GIT_WORK_TREE, but not GIT_EDITOR). Only the names of the variables are listed, not their value, even if they are set.

--git-dir

Show `$GIT_DIR` if defined. Otherwise show the path to the .git directory. The path shown, when relative, is relative to the current working directory.

If `$GIT_DIR` is not defined and the current directory is not detected to lie in a Git repository or work tree print a message to stderr and exit with nonzero status.

--git-common-dir

Show `$GIT_COMMON_DIR` if defined, else `$GIT_DIR`.

--is-inside-git-dir

When the current working directory is below the repository directory print "true", otherwise "false".

--is-inside-work-tree

When the current working directory is inside the work tree of the repository print "true", otherwise "false".

--is-bare-repository

When the repository is bare print "true", otherwise "false".

--resolve-git-dir <path>

Check if <path> is a valid repository or a gitfile that points at a valid repository, and print the location of the repository. If <path> is a gitfile then the resolved path to the real repository is printed.

--git-path <path>

Resolve "$GIT_DIR/<path>" and takes other path relocation variables such as $GIT_OBJECT_DIRECTORY, $GIT_INDEX_FILE… into account. For example, if $GIT_OBJECT_DIRECTORY is set to /foo/bar then "git rev-parse --git-path objects/abc"

returns /foo/bar/abc.

--show-cdup

When the command is invoked from a subdirectory, show the path of the top-level directory relative to the current directory (typically a sequence of "../", or an empty string).

--show-prefix

When the command is invoked from a subdirectory, show the path of the current directory relative to the top-level directory.

--show-toplevel

Show the absolute path of the top-level directory.

--shared-index-path

Show the path to the shared index file in split index mode, or empty if not in split-index mode.

Other Options

--since=datestring

--after=datestring

Parse the date string, and output the corresponding --max-age= parameter for *git rev-list*.

--until=datestring

--before=datestring

Parse the date string, and output the corresponding --min-age= parameter for *git rev-list*.

<args>…

Flags and parameters to be parsed.

SPECIFYING REVISIONS

A revision parameter *<rev>* typically, but not necessarily, names a commit object. It uses what is called an *extended SHA-1* syntax. Here are various ways to spell object names. The ones listed near the end of this list name trees and blobs contained in a commit.

<sha1>, e.g. *dae86e1950b1277e545cee180551750029cfe735*, *dae86e*

The full SHA-1 object name (40-byte hexadecimal string), or a leading substring that is unique within the repository. E.g. dae86e1950b1277e545cee180551750029cfe735 and dae86e both name the same commit object if there is no other object in your repository whose object name starts with dae86e.

<describeOutput>, e.g. *v1.7.4.2-679-g3bee7fb*

Output from `git describe`; i.e. a closest tag, optionally followed by a dash and a number of commits, followed by a dash, a *g*, and an abbreviated object name.

<refname>, e.g. *master*, *heads/master*, *refs/heads/master*

A symbolic ref name. E.g. *master* typically means the commit object referenced by *refs/heads/master*. If you happen to have both *heads/master* and *tags/master*, you can explicitly say *heads/master* to tell Git which one you mean. When ambiguous, a *<refname>* is disambiguated by taking the first match in the following rules:

1. If *$GIT_DIR/<refname>* exists, that is what you mean (this is usually useful only for *HEAD*, *FETCH_HEAD*, *ORIG_HEAD*, *MERGE_HEAD* and *CHERRY_PICK_HEAD*);
2. otherwise, *refs/<refname>* if it exists;
3. otherwise, *refs/tags/<refname>* if it exists;
4. otherwise, *refs/heads/<refname>* if it exists;
5. otherwise, *refs/remotes/<refname>* if it exists;
6. otherwise, *refs/remotes/<refname>/HEAD* if it exists.

   HEAD names the commit on which you based the changes in the working tree. *FETCH_HEAD* records the branch which you fetched from a remote repository with your last `git fetch` invocation. *ORIG_HEAD* is created by commands that move your *HEAD* in a drastic way, to record the position of the *HEAD* before their operation, so that you can easily change the tip of the branch back to the state before you ran them. *MERGE_HEAD* records the commit(s) which you are merging into your branch when you run `git merge`. *CHERRY_PICK_HEAD* records the commit which you are cherry-picking when you run `git cherry-pick`.

   Note that any of the *refs/** cases above may come either from the *$GIT_DIR/refs* directory or from the *$GIT_DIR/packed-refs* file. While the ref name encoding is unspecified, UTF-8 is preferred as some output processing may assume ref names in UTF-8.

@

@ alone is a shortcut for *HEAD*.

<refname>@{<date>}, e.g. *master@{yesterday}*, *HEAD@{5 minutes ago}*

A ref followed by the suffix @ with a date specification enclosed in a brace pair (e.g. *{yesterday}*, *{1 month 2 weeks 3 days 1 hour 1 second ago}* or *{1979-02-26 18:30:00}*) specifies the value of the ref at a prior point in time. This suffix may only be used immediately following a ref name and the ref must have an existing log (*$GIT_DIR/logs/<ref>*). Note that this looks up the state of your **local** ref at a given time; e.g., what was in your local *master* branch last week. If you want to look at commits made during certain times, see *--since* and *--until*.

<refname>@{<n>}, e.g. *master@{1}*

A ref followed by the suffix @ with an ordinal specification enclosed in a brace pair (e.g. *{1}*, *{15}*) specifies the n-th prior value of that ref. For example *master@{1}* is the immediate prior value of *master* while *master@{5}* is the 5th prior value of *master*. This suffix may only be used immediately following a ref name and the ref must have an existing log (*$GIT_DIR/logs/<refname>*).

@{<n>}, e.g. *@{1}*

You can use the @ construct with an empty ref part to get at a reflog entry of the current branch. For example, if you are on branch *blabla* then *@{1}* means the same as *blabla@{1}*.

@{-<n>}, e.g. *@{-1}*

The construct *@{-<n>}* means the <n>th branch/commit checked out before the current one.

<branchname>@{upstream}, e.g. *master@{upstream}*, *@{u}*

The suffix *@{upstream}* to a branchname (short form *<branchname>@{u}*) refers to the branch that the branch specified by branchname is set to build on top of (configured with `branch.<name>.remote` and `branch.<name>.merge`). A missing branchname defaults to the current one.

<branchname>@{push}, e.g. *master@{push}*, *@{push}*

The suffix *@{push}* reports the branch "where we would push to" if `git push` were run while `branchname` was checked out (or the current *HEAD* if no branchname is specified). Since our push destination is in a remote repository, of course, we report the local tracking branch that corresponds to that branch (i.e., something in *refs/remotes/*).

Here’s an example to make it more clear:

```
$ git config push.default current
$ git config remote.pushdefault myfork
$ git checkout -b mybranch origin/master

$ git rev-parse --symbolic-full-name @{upstream}
refs/remotes/origin/master

$ git rev-parse --symbolic-full-name @{push}
refs/remotes/myfork/mybranch
```

Note in the example that we set up a triangular workflow, where we pull from one location and push to another. In a non-triangular workflow, *@{push}* is the same as *@{upstream}*, and there is no need for it.

<rev>^, e.g. *HEAD^, v1.5.1^0*

A suffix ^ to a revision parameter means the first parent of that commit object. ^*<n>* means the <n>th parent (i.e. *<rev>*^ is equivalent to *<rev>^1*). As a special rule, *<rev>^0* means the commit itself and is used when *<rev>* is the object name of a tag object that refers to a commit object.

<rev>~<n>, e.g. *master~3*

A suffix ~*<n>* to a revision parameter means the commit object that is the <n>th generation ancestor of the named commit object, following only the first parents. I.e. *<rev>~3* is equivalent to *<rev>*^^^ which is equivalent to *<rev>^1^1^1*. See below for an illustration of the usage of this form.

<rev>^{<type>}, e.g. *v0.99.8^{commit}*

A suffix ^ followed by an object type name enclosed in brace pair means dereference the object at *<rev>* recursively until an object of type *<type>* is found or the object cannot be dereferenced anymore (in which case, barf). For example, if *<rev>* is a commit-ish, *<rev>^{commit}* describes the corresponding commit object. Similarly, if *<rev>* is a tree-ish, *<rev>^{tree}* describes the corresponding tree object. *<rev>^0* is a short-hand for *<rev>^{commit}*.

rev^{object} can be used to make sure *rev* names an object that exists, without requiring *rev* to be a tag, and without dereferencing *rev*; because a tag is already an object, it does not have to be dereferenced even once to get to an object.

rev^{tag} can be used to ensure that *rev* identifies an existing tag object.

<rev>^{}, e.g. *v0.99.8^{}*

A suffix ^ followed by an empty brace pair means the object could be a tag, and dereference the tag recursively until a non-tag object is found.

<rev>^{/<text>}, e.g. *HEAD^{/fix nasty bug}*

A suffix ^ to a revision parameter, followed by a brace pair that contains a text led by a slash, is the same as the *:/fix nasty bug* syntax below except that it returns the youngest matching commit which is reachable from the *<rev>* before ^.

:/<text>, e.g. *:/fix nasty bug*

A colon, followed by a slash, followed by a text, names a commit whose commit message matches the specified regular expression. This name returns the youngest matching commit which is reachable from any ref. The regular expression can match any part of the commit message. To match messages starting with a string, one can use e.g. *:/^foo*. The special sequence *:/!* is reserved for modifiers to what is matched. *:/!-foo* performs a negative match, while *:/!!foo* matches a literal *!* character, followed by *foo*. Any other sequence beginning with *:/!* is reserved for now.

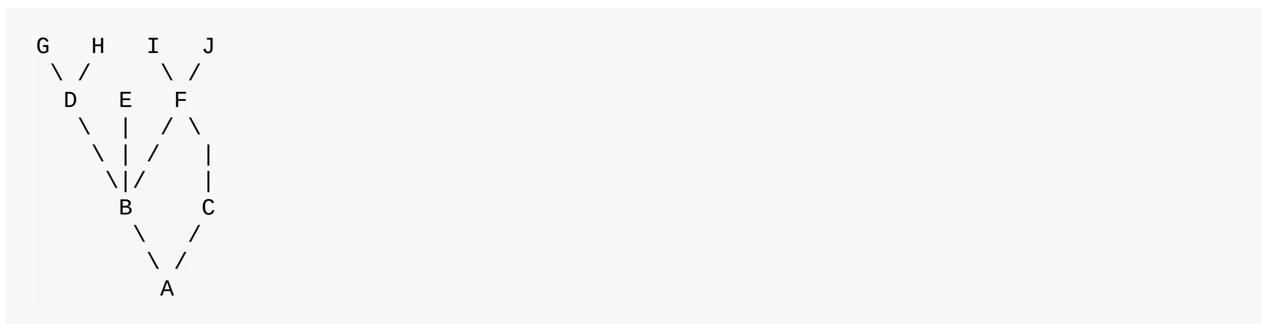
<rev>:<path>, e.g. *HEAD:README*, *:README*, *master:./README*

A suffix *:* followed by a path names the blob or tree at the given path in the tree-ish object named by the part before the colon. *:path* (with an empty part before the colon) is a special case of the syntax described next: content recorded in the index at the given path. A path starting with *./* or *../* is relative to the current working directory. The given path will be converted to be relative to the working tree's root directory. This is most useful to address a blob or tree from a commit or tree that has the same tree structure as the working tree.

:<n>:<path>, e.g. *:0:README*, *:README*

A colon, optionally followed by a stage number (0 to 3) and a colon, followed by a path, names a blob object in the index at the given path. A missing stage number (and the colon that follows it) names a stage 0 entry. During a merge, stage 1 is the common ancestor, stage 2 is the target branch's version (typically the current branch), and stage 3 is the version from the branch which is being merged.

Here is an illustration, by Jon Loeliger. Both commit nodes B and C are parents of commit node A. Parent commits are ordered left-to-right.



```
G   H   I   J
 \ /     \ /
  D   E   F
   \  |  / \
    \ | /   |
     \|/    |
      B     C
       \   /
        \ /
         A
```

```
A =      = A^0
B = A^   = A^1     = A~1
C = A^2  = A^2
D = A^^  = A^1^1   = A~2
E = B^2  = A^^2
F = B^3  = A^^3
G = A^^^ = A^1^1^1 = A~3
H = D^2  = B^^2    = A^^^2  = A~2^2
I = F^   = B^3^    = A^^3^
J = F^2  = B^3^2   = A^^3^2
```

SPECIFYING RANGES

History traversing commands such as `git log` operate on a set of commits, not just a single commit. To these commands, specifying a single revision with the notation described in the previous section means the set of commits reachable from that commit, following the commit ancestry chain.

To exclude commits reachable from a commit, a prefix ^ notation is used. E.g. *^r1 r2* means commits reachable from *r2* but exclude the ones reachable from *r1*.

This set operation appears so often that there is a shorthand for it. When you have two commits *r1* and *r2* (named according to the syntax explained in SPECIFYING REVISIONS above), you can ask for commits that are reachable from r2 excluding those that are reachable from r1 by *^r1 r2* and it can be written as *r1..r2*.

A similar notation *r1...r2* is called symmetric difference of *r1* and *r2* and is defined as *r1 r2 --not $(git merge-base --all r1 r2)*. It is the set of commits that are reachable from either one of *r1* or *r2* but not from both.

In these two shorthands, you can omit one end and let it default to HEAD. For example, *origin..* is a shorthand for *origin..HEAD* and asks "What did I do since I forked from the origin branch?" Similarly, *..origin* is a shorthand for *HEAD..origin* and asks "What did the origin do since I forked from them?" Note that *..* would mean *HEAD..HEAD* which is an empty range that is both reachable and unreachable from HEAD.

Two other shorthands for naming a set that is formed by a commit and its parent commits exist. The *r1^@* notation means all parents of *r1*. *r1^!* includes commit *r1* but excludes all of its parents.

To summarize:

<rev>

Include commits that are reachable from (i.e. ancestors of) <rev>.

^<rev>

Exclude commits that are reachable from (i.e. ancestors of) <rev>.

<rev1>..<rev2>

Include commits that are reachable from <rev2> but exclude those that are reachable from <rev1>. When either <rev1> or <rev2> is omitted, it defaults to *HEAD*.

<rev1>...<rev2>

Include commits that are reachable from either <rev1> or <rev2> but exclude those that are reachable from both. When either <rev1> or <rev2> is omitted, it defaults to *HEAD*.

<rev>^@, e.g. *HEAD^@*

A suffix ^ followed by an at sign is the same as listing all parents of *<rev>* (meaning, include anything reachable from its parents, but not the commit itself).

<rev>^!, e.g. *HEAD^!*

A suffix ^ followed by an exclamation mark is the same as giving commit *<rev>* and then all its parents prefixed with ^ to exclude them (and their ancestors).

Here are a handful of examples:

```
D                G H D
D F              G H I J D F
^G D             H D
^D B             E I J F B
B..C             C
B...C            G H D E B C
^D B C           E I J F B C
C                I J F C
C^@              I J F
C^!              C
F^! D            G H D F
```

PARSEOPT

In `--parseopt` mode, *git rev-parse* helps massaging options to bring to shell scripts the same facilities C builtins have. It works as an option normalizer (e.g. splits single switches aggregate values), a bit like `getopt(1)` does.

It takes on the standard input the specification of the options to parse and understand, and echoes on the standard output a string suitable for `sh(1)` `eval` to replace the arguments with normalized ones. In case of error, it outputs usage on the standard error stream, and exits with code 129.

Note: Make sure you quote the result when passing it to `eval`. See below for an example.

Input Format

git rev-parse --parseopt input format is fully text based. It has two parts, separated by a line that contains only `--`. The lines before the separator (should be one or more) are used for the usage. The lines after the separator describe the options.

Each line of options has this format:

```
<opt-spec><flags>*<arg-hint>? SP+ help LF
```

`<opt-spec>`

its format is the short option character, then the long option name separated by a comma. Both parts are not required, though at least one is necessary. May not contain any of the `<flags>` characters. `h,help`, `dry-run` and `f` are examples of correct `<opt-spec>`.

`<flags>`

`<flags>` are of `*`, `=`, `?` or `!`.

- Use `=` if the option takes an argument.
- Use `?` to mean that the option takes an optional argument. You probably want to use the `--stuck-long` mode to be able to unambiguously parse the optional argument.
- Use `*` to mean that this option should not be listed in the usage generated for the `-h` argument. It's shown for `--help-all` as documented in gitcli[7].
- Use `!` to not make the corresponding negated long option available.

`<arg-hint>`

`<arg-hint>`, if specified, is used as a name of the argument in the help output, for options that take arguments. `<arg-hint>` is terminated by the first whitespace. It is customary to use a dash to separate words in a multi-word argument hint.

The remainder of the line, after stripping the spaces, is used as the help associated to the option.

Blank lines are ignored, and lines that don't match this specification are used as option group headers (start the line with a space to create such lines on purpose).

Example

```
OPTS_SPEC="\
some-command [options] <args>...

some-command does foo and bar!
--
h,help    show the help

foo       some nifty option --foo
bar=      some cool option --bar with an argument
baz=arg   another cool option --baz with a named argument
qux?path  qux may take a path argument but has meaning by itself

  An option group Header
C?        option C with an optional argument"

eval "$(echo "$OPTS_SPEC" | git rev-parse --parseopt -- "$@" || echo exit $?)"
```

Usage text

When `"$@"` is `-h` or `--help` in the above example, the following usage text would be shown:

```
usage: some-command [options] <args>...

    some-command does foo and bar!

    -h, --help            show the help
    --foo                 some nifty option --foo
    --bar ...             some cool option --bar with an argument
    --baz <arg>           another cool option --baz with a named argument
    --qux[=<path>]        qux may take a path argument but has meaning by itself

An option group Header
    -C[...]               option C with an optional argument
```

SQ-QUOTE

In `--sq-quote` mode, *git rev-parse* echoes on the standard output a single line suitable for `sh(1)` `eval`. This line is made by normalizing the arguments following `--sq-quote`. Nothing other than quoting the arguments is done.

If you want command input to still be interpreted as usual by *git rev-parse* before the output is shell quoted, see the `--sq` option.

Example

```
$ cat >your-git-script.sh <<\EOF
#!/bin/sh
args=$(git rev-parse --sq-quote "$@")   # quote user-supplied arguments
command="git frotz -n24 $args"          # and use it inside a handcrafted
                    # command line
eval "$command"
EOF

$ sh your-git-script.sh "a b'c"
```

EXAMPLES

- Print the object name of the current commit:

  ```
  $ git rev-parse --verify HEAD
  ```

- Print the commit object name from the revision in the $REV shell variable:

  ```
  $ git rev-parse --verify $REV^{commit}
  ```

  This will error out if $REV is empty or not a valid revision.

- Similar to above:

  ```
  $ git rev-parse --default master --verify $REV
  ```

  but if $REV is empty, the commit object name from master will be printed.

GIT

Part of the git[1] suite

show-ref

NAME

git-show-ref - List references in a local repository

SYNOPSIS

```
git show-ref [-q|--quiet] [--verify] [--head] [-d|--dereference]
         [-s|--hash[=<n>]] [--abbrev[=<n>]] [--tags]
         [--heads] [--] [<pattern>…]
git show-ref --exclude-existing[=<pattern>]
```

DESCRIPTION

Displays references available in a local repository along with the associated commit IDs. Results can be filtered using a pattern and tags can be dereferenced into object IDs. Additionally, it can be used to test whether a particular ref exists.

By default, shows the tags, heads, and remote refs.

The --exclude-existing form is a filter that does the inverse. It reads refs from stdin, one ref per line, and shows those that don’t exist in the local repository.

Use of this utility is encouraged in favor of directly accessing files under the `.git` directory.

OPTIONS

--head

Show the HEAD reference, even if it would normally be filtered out.

--tags

--heads

Limit to "refs/heads" and "refs/tags", respectively. These options are not mutually exclusive; when given both, references stored in "refs/heads" and "refs/tags" are displayed.

-d

--dereference

Dereference tags into object IDs as well. They will be shown with "^{}" appended.

-s

--hash[=<n>]

Only show the SHA-1 hash, not the reference name. When combined with --dereference the dereferenced tag will still be shown after the SHA-1.

--verify

Enable stricter reference checking by requiring an exact ref path. Aside from returning an error code of 1, it will also print an error message if *--quiet* was not specified.

--abbrev[=<n>]

Abbreviate the object name. When using `--hash`, you do not have to say `--hash --abbrev`; `--hash=n` would do.

-q

--quiet

Do not print any results to stdout. When combined with *--verify* this can be used to silently check if a reference exists.

--exclude-existing[=<pattern>]

Make *git show-ref* act as a filter that reads refs from stdin of the form "`^(?:<anything>\s)?<refname>(?:\^{})?$`" and performs the following actions on each: (1) strip "^{}" at the end of line if any; (2) ignore if pattern is provided and does not head-match refname; (3) warn if refname is not a well-formed refname and skip; (4) ignore if refname is a ref that exists in the local repository; (5) otherwise output the line.

<pattern>…

Show references matching one or more patterns. Patterns are matched from the end of the full name, and only complete parts are matched, e.g. *master* matches *refs/heads/master*, *refs/remotes/origin/master*, *refs/tags/jedi/master* but not *refs/heads/mymaster* or *refs/remotes/master/jedi*.

OUTPUT

The output is in the format: *<SHA-1 ID>* *<space>* *<reference name>*.

```
$ git show-ref --head --dereference
832e76a9899f560a90ffd62ae2ce83bbeff58f54 HEAD
832e76a9899f560a90ffd62ae2ce83bbeff58f54 refs/heads/master
832e76a9899f560a90ffd62ae2ce83bbeff58f54 refs/heads/origin
3521017556c5de4159da4615a39fa4d5d2c279b5 refs/tags/v0.99.9c
6ddc0964034342519a87fe013781abf31c6db6ad refs/tags/v0.99.9c^{}
055e4ae3ae6eb344cbabf2a5256a49ea66040131 refs/tags/v1.0rc4
423325a2d24638ddcc82ce47be5e40be550f4507 refs/tags/v1.0rc4^{}
...
```

When using --hash (and not --dereference) the output format is: *<SHA-1 ID>*

```
$ git show-ref --heads --hash
2e3ba0114a1f52b47df29743d6915d056be13278
185008ae97960c8d551adcd9e23565194651b5d1
03adf42c988195b50e1a1935ba5fcbc39b2b029b
...
```

EXAMPLE

To show all references called "master", whether tags or heads or anything else, and regardless of how deep in the reference naming hierarchy they are, use:

```
        git show-ref master
```

This will show "refs/heads/master" but also "refs/remote/other-repo/master", if such references exists.

When using the *--verify* flag, the command requires an exact path:

```
        git show-ref --verify refs/heads/master
```

will only match the exact branch called "master".

If nothing matches, *git show-ref* will return an error code of 1, and in the case of verification, it will show an error message.

For scripting, you can ask it to be quiet with the "--quiet" flag, which allows you to do things like

```
        git show-ref --quiet --verify -- "refs/heads/$headname" ||
            echo "$headname is not a valid branch"
```

to check whether a particular branch exists or not (notice how we don’t actually want to show any results, and we want to use the full refname for it in order to not trigger the problem with ambiguous partial matches).

To show only tags, or only proper branch heads, use "--tags" and/or "--heads" respectively (using both means that it shows tags and heads, but not other random references under the refs/ subdirectory).

To do automatic tag object dereferencing, use the "-d" or "--dereference" flag, so you can do

```
git show-ref --tags --dereference
```

to get a listing of all tags together with what they dereference.

FILES

`.git/refs/*`, `.git/packed-refs`

SEE ALSO

git-for-each-ref[1], git-ls-remote[1], git-update-ref[1], gitrepository-layout[5]

GIT

Part of the git[1] suite

symbolic-ref

NAME

git-symbolic-ref - Read, modify and delete symbolic refs

SYNOPSIS

```
git symbolic-ref [-m <reason>] <name> <ref>
git symbolic-ref [-q] [--short] <name>
git symbolic-ref --delete [-q] <name>
```

DESCRIPTION

Given one argument, reads which branch head the given symbolic ref refers to and outputs its path, relative to the `.git/` directory. Typically you would give `HEAD` as the <name> argument to see which branch your working tree is on.

Given two arguments, creates or updates a symbolic ref <name> to point at the given branch <ref>.

Given `--delete` and an additional argument, deletes the given symbolic ref.

A symbolic ref is a regular file that stores a string that begins with `ref: refs/`. For example, your `.git/HEAD` is a regular file whose contents is `ref: refs/heads/master`.

OPTIONS

-d

--delete

Delete the symbolic ref <name>.

-q

--quiet

Do not issue an error message if the <name> is not a symbolic ref but a detached HEAD; instead exit with non-zero status silently.

--short

When showing the value of <name> as a symbolic ref, try to shorten the value, e.g. from `refs/heads/master` to `master`.

-m

Update the reflog for <name> with <reason>. This is valid only when creating or updating a symbolic ref.

NOTES

In the past, `.git/HEAD` was a symbolic link pointing at `refs/heads/master`. When we wanted to switch to another branch, we did `ln -sf refs/heads/newbranch .git/HEAD`, and when we wanted to find out which branch we are on, we did `readlink .git/HEAD`. But symbolic links are not entirely portable, so they are now deprecated and symbolic refs (as described above) are used by default.

git symbolic-ref will exit with status 0 if the contents of the symbolic ref were printed correctly, with status 1 if the requested name is not a symbolic ref, or 128 if another error occurs.

GIT

Part of the git[1] suite

update-index

NAME

git-update-index - Register file contents in the working tree to the index

SYNOPSIS

```
git update-index
         [--add] [--remove | --force-remove] [--replace]
         [--refresh] [-q] [--unmerged] [--ignore-missing]
         [(--cacheinfo <mode>,<object>,<file>)…]
         [--chmod=(+|-)x]
         [--[no-]assume-unchanged]
         [--[no-]skip-worktree]
         [--ignore-submodules]
         [--[no-]split-index]
         [--[no-|test-|force-]untracked-cache]
         [--really-refresh] [--unresolve] [--again | -g]
         [--info-only] [--index-info]
         [-z] [--stdin] [--index-version <n>]
         [--verbose]
         [--] [<file>…]
```

DESCRIPTION

Modifies the index or directory cache. Each file mentioned is updated into the index and any *unmerged* or *needs updating* state is cleared.

See also git-add[1] for a more user-friendly way to do some of the most common operations on the index.

The way *git update-index* handles files it is told about can be modified using the various options:

OPTIONS

--add

If a specified file isn't in the index already then it's added. Default behaviour is to ignore new files.

--remove

If a specified file is in the index but is missing then it’s removed. Default behavior is to ignore removed file.

--refresh

Looks at the current index and checks to see if merges or updates are needed by checking stat() information.

-q

Quiet. If --refresh finds that the index needs an update, the default behavior is to error out. This option makes *git update-index* continue anyway.

--ignore-submodules

Do not try to update submodules. This option is only respected when passed before --refresh.

--unmerged

If --refresh finds unmerged changes in the index, the default behavior is to error out. This option makes *git update-index* continue anyway.

--ignore-missing

Ignores missing files during a --refresh

--cacheinfo <mode>,<object>,<path>

--cacheinfo <mode> <object> <path>

Directly insert the specified info into the index. For backward compatibility, you can also give these three arguments as three separate parameters, but new users are encouraged to use a single-parameter form.

--index-info

Read index information from stdin.

--chmod=(+|-)x

Set the execute permissions on the updated files.

--[no-]assume-unchanged

When this flag is specified, the object names recorded for the paths are not updated. Instead, this option sets/unsets the "assume unchanged" bit for the paths. When the "assume unchanged" bit is on, the user promises not to change the file and allows Git to assume that the working tree file matches what is recorded in the index. If you want to

change the working tree file, you need to unset the bit to tell Git. This is sometimes helpful when working with a big project on a filesystem that has very slow lstat(2) system call (e.g. cifs).

Git will fail (gracefully) in case it needs to modify this file in the index e.g. when merging in a commit; thus, in case the assumed-untracked file is changed upstream, you will need to handle the situation manually.

--really-refresh

Like *--refresh*, but checks stat information unconditionally, without regard to the "assume unchanged" setting.

--[no-]skip-worktree

When one of these flags is specified, the object name recorded for the paths are not updated. Instead, these options set and unset the "skip-worktree" bit for the paths. See section "Skip-worktree bit" below for more information.

-g

--again

Runs *git update-index* itself on the paths whose index entries are different from those from the `HEAD` commit.

--unresolve

Restores the *unmerged* or *needs updating* state of a file during a merge if it was cleared by accident.

--info-only

Do not create objects in the object database for all <file> arguments that follow this flag; just insert their object IDs into the index.

--force-remove

Remove the file from the index even when the working directory still has such a file. (Implies --remove.)

--replace

By default, when a file `path` exists in the index, *git update-index* refuses an attempt to add `path/file`. Similarly if a file `path/file` exists, a file `path` cannot be added. With --replace flag, existing entries that conflict with the entry being added are automatically removed with warning messages.

--stdin

Instead of taking list of paths from the command line, read list of paths from the standard input. Paths are separated by LF (i.e. one path per line) by default.

--verbose

Report what is being added and removed from index.

--index-version <n>

Write the resulting index out in the named on-disk format version. Supported versions are 2, 3 and 4. The current default version is 2 or 3, depending on whether extra features are used, such as `git add -N`.

Version 4 performs a simple pathname compression that reduces index size by 30%-50% on large repositories, which results in faster load time. Version 4 is relatively young (first released in in 1.8.0 in October 2012). Other Git implementations such as JGit and libgit2 may not support it yet.

-z

Only meaningful with `--stdin` or `--index-info`; paths are separated with NUL character instead of LF.

--split-index

--no-split-index

Enable or disable split index mode. If enabled, the index is split into two files, $GIT_DIR/index and $GIT_DIR/sharedindex.<SHA-1>. Changes are accumulated in $GIT_DIR/index while the shared index file contains all index entries stays unchanged. If split-index mode is already enabled and `--split-index` is given again, all changes in $GIT_DIR/index are pushed back to the shared index file. This mode is designed for very large indexes that take a significant amount of time to read or write.

--untracked-cache

--no-untracked-cache

Enable or disable untracked cache feature. Please use `--test-untracked-cache` before enabling it.

These options take effect whatever the value of the `core.untrackedCache` configuration variable (see git-config[1]). But a warning is emitted when the change goes against the configured value, as the configured value will take effect next time the index is read and this will remove the intended effect of the option.

--test-untracked-cache

Only perform tests on the working directory to make sure untracked cache can be used. You have to manually enable untracked cache using `--untracked-cache` or `--force-untracked-cache` or the `core.untrackedCache` configuration variable afterwards if you really want to use it. If a test fails the exit code is 1 and a message explains what is not working as needed, otherwise the exit code is 0 and OK is printed.

--force-untracked-cache

Same as `--untracked-cache`. Provided for backwards compatibility with older versions of Git where `--untracked-cache` used to imply `--test-untracked-cache` but this option would enable the extension unconditionally.

--

Do not interpret any more arguments as options.

<file>

Files to act on. Note that files beginning with *.* are discarded. This includes `./file` and `dir/./file`. If you don't want this, then use cleaner names. The same applies to directories ending */* and paths with *//*

Using --refresh

--refresh does not calculate a new sha1 file or bring the index up-to-date for mode/content changes. But what it **does** do is to "re-match" the stat information of a file with the index, so that you can refresh the index for a file that hasn't been changed but where the stat entry is out of date.

For example, you'd want to do this after doing a *git read-tree*, to link up the stat index details with the proper files.

Using --cacheinfo or --info-only

--cacheinfo is used to register a file that is not in the current working directory. This is useful for minimum-checkout merging.

To pretend you have a file with mode and sha1 at path, say:

```
$ git update-index --cacheinfo <mode>,<sha1>,<path>
```

--info-only is used to register files without placing them in the object database. This is useful for status-only repositories.

Both *--cacheinfo* and *--info-only* behave similarly: the index is updated but the object database isn't. *--cacheinfo* is useful when the object is in the database but the file isn't available locally. *--info-only* is useful when the file is available, but you do not wish to update the object database.

Using --index-info

`--index-info` is a more powerful mechanism that lets you feed multiple entry definitions from the standard input, and designed specifically for scripts. It can take inputs of three formats:

1. mode SP sha1 TAB path

   The first format is what "git-apply --index-info" reports, and used to reconstruct a partial tree that is used for phony merge base tree when falling back on 3-way merge.

2. mode SP type SP sha1 TAB path

   The second format is to stuff *git ls-tree* output into the index file.

3. mode SP sha1 SP stage TAB path

   This format is to put higher order stages into the index file and matches *git ls-files --stage* output.

To place a higher stage entry to the index, the path should first be removed by feeding a mode=0 entry for the path, and then feeding necessary input lines in the third format.

For example, starting with this index:

```
$ git ls-files -s
100644 8a1218a1024a212bb3db30becd860315f9f3ac52 0	frotz
```

you can feed the following input to `--index-info`:

```
$ git update-index --index-info
0 0000000000000000000000000000000000000000	frotz
100644 8a1218a1024a212bb3db30becd860315f9f3ac52 1	frotz
100755 8a1218a1024a212bb3db30becd860315f9f3ac52 2	frotz
```

The first line of the input feeds 0 as the mode to remove the path; the SHA-1 does not matter as long as it is well formatted. Then the second and third line feeds stage 1 and stage 2 entries for that path. After the above, we would end up with this:

```
$ git ls-files -s
100644 8a1218a1024a212bb3db30becd860315f9f3ac52 1	frotz
100755 8a1218a1024a212bb3db30becd860315f9f3ac52 2	frotz
```

Using “assume unchanged” bit

Many operations in Git depend on your filesystem to have an efficient `lstat(2)` implementation, so that `st_mtime` information for working tree files can be cheaply checked to see if the file contents have changed from the version recorded in the index file. Unfortunately, some filesystems have inefficient `lstat(2)`. If your filesystem is one of them, you can set "assume unchanged" bit to paths you have not changed to cause Git not to do this check. Note that setting this bit on a path does not mean Git will check the contents of the file to see if it has changed — it makes Git to omit any checking and assume it has **not** changed. When you make changes to working tree files, you have to explicitly tell Git about it by dropping "assume unchanged" bit, either before or after you modify them.

In order to set "assume unchanged" bit, use `--assume-unchanged` option. To unset, use `--no-assume-unchanged`. To see which files have the "assume unchanged" bit set, use `git ls-files -v` (see git-ls-files[1]).

The command looks at `core.ignorestat` configuration variable. When this is true, paths updated with `git update-index paths...` and paths updated with other Git commands that update both index and working tree (e.g. *git apply --index*, *git checkout-index -u*, and *git read-tree -u*) are automatically marked as "assume unchanged". Note that "assume unchanged" bit is **not** set if `git update-index --refresh` finds the working tree file matches the index (use `git update-index --really-refresh` if you want to mark them as "assume unchanged").

Examples

To update and refresh only the files already checked out:

```
$ git checkout-index -n -f -a && git update-index --ignore-missing --refresh
```

On an inefficient filesystem with `core.ignorestat` set

```
$ git update-index --really-refresh              (1)
$ git update-index --no-assume-unchanged foo.c   (2)
$ git diff --name-only                           (3)
$ edit foo.c
$ git diff --name-only                           (4)
M foo.c
$ git update-index foo.c                         (5)
$ git diff --name-only                           (6)
$ edit foo.c
$ git diff --name-only                           (7)
$ git update-index --no-assume-unchanged foo.c   (8)
$ git diff --name-only                           (9)
M foo.c
```

1. forces lstat(2) to set "assume unchanged" bits for paths that match index.
2. mark the path to be edited.
3. this does lstat(2) and finds index matches the path.
4. this does lstat(2) and finds index does **not** match the path.
5. registering the new version to index sets "assume unchanged" bit.
6. and it is assumed unchanged.
7. even after you edit it.
8. you can tell about the change after the fact.
9. now it checks with lstat(2) and finds it has been changed.

Skip-worktree bit

Skip-worktree bit can be defined in one (long) sentence: When reading an entry, if it is marked as skip-worktree, then Git pretends its working directory version is up to date and read the index version instead.

To elaborate, "reading" means checking for file existence, reading file attributes or file content. The working directory version may be present or absent. If present, its content may match against the index version or not. Writing is not affected by this bit, content safety is still first priority. Note that Git *can* update working directory file, that is marked skip-worktree, if it is safe to do so (i.e. working directory version matches index version)

Although this bit looks similar to assume-unchanged bit, its goal is different from assume-unchanged bit's. Skip-worktree also takes precedence over assume-unchanged bit when both are set.

Untracked cache

This cache is meant to speed up commands that involve determining untracked files such as `git status`.

This feature works by recording the mtime of the working tree directories and then omitting reading directories and stat calls against files in those directories whose mtime hasn't changed. For this to work the underlying operating system and file system must change the `st_mtime` field of directories if files in the directory are added, modified or deleted.

You can test whether the filesystem supports that with the `--test-untracked-cache` option. The `--untracked-cache` option used to implicitly perform that test in older versions of Git, but that's no longer the case.

If you want to enable (or disable) this feature, it is easier to use the `core.untrackedCache` configuration variable (see git-config[1]) than using the `--untracked-cache` option to `git update-index` in each repository, especially if you want to do so across all repositories you use, because you can set the configuration variable to `true` (or `false`) in your `$HOME/.gitconfig` just once and have it affect all repositories you touch.

When the `core.untrackedCache` configuration variable is changed, the untracked cache is added to or removed from the index the next time a command reads the index; while when `--[no-|force-]untracked-cache` are used, the untracked cache is immediately added to or removed from the index.

Configuration

The command honors `core.filemode` configuration variable. If your repository is on a filesystem whose executable bits are unreliable, this should be set to *false* (see git-config[1]). This causes the command to ignore differences in file modes recorded in the index and the file mode on the filesystem if they differ only on executable bit. On such an unfortunate filesystem, you may need to use *git update-index --chmod=*.

Quite similarly, if `core.symlinks` configuration variable is set to *false* (see git-config[1]), symbolic links are checked out as plain files, and this command does not modify a recorded file mode from symbolic link to regular file.

The command looks at `core.ignorestat` configuration variable. See *Using "assume unchanged" bit* section above.

The command also looks at `core.trustctime` configuration variable. It can be useful when the inode change time is regularly modified by something outside Git (file system crawlers and backup systems use ctime for marking files processed) (see git-config[1]).

The untracked cache extension can be enabled by the `core.untrackedCache` configuration variable (see git-config[1]).

SEE ALSO

git-config[1], git-add[1], git-ls-files[1]

GIT

Part of the git[1] suite

update-ref

NAME

git-update-ref - Update the object name stored in a ref safely

SYNOPSIS

```
git update-ref [-m <reason>] (-d <ref> [<oldvalue>] | [--no-deref] [--create-reflog] <ref
```

DESCRIPTION

Given two arguments, stores the <newvalue> in the <ref>, possibly dereferencing the symbolic refs. E.g. `git update-ref HEAD <newvalue>` updates the current branch head to the new object.

Given three arguments, stores the <newvalue> in the <ref>, possibly dereferencing the symbolic refs, after verifying that the current value of the <ref> matches <oldvalue>. E.g. `git update-ref refs/heads/master <newvalue> <oldvalue>` updates the master branch head to <newvalue> only if its current value is <oldvalue>. You can specify 40 "0" or an empty string as <oldvalue> to make sure that the ref you are creating does not exist.

It also allows a "ref" file to be a symbolic pointer to another ref file by starting with the four-byte header sequence of "ref:".

More importantly, it allows the update of a ref file to follow these symbolic pointers, whether they are symlinks or these "regular file symbolic refs". It follows **real** symlinks only if they start with "refs/": otherwise it will just try to read them and update them as a regular file (i.e. it will allow the filesystem to follow them, but will overwrite such a symlink to somewhere else with a regular filename).

If --no-deref is given, <ref> itself is overwritten, rather than the result of following the symbolic pointers.

In general, using

```
git update-ref HEAD "$head"
```

should be a *lot* safer than doing

```
echo "$head" > "$GIT_DIR/HEAD"
```

both from a symlink following standpoint **and** an error checking standpoint. The "refs/" rule for symlinks means that symlinks that point to "outside" the tree are safe: they'll be followed for reading but not for writing (so we'll never write through a ref symlink to some other tree, if you have copied a whole archive by creating a symlink tree).

With `-d` flag, it deletes the named <ref> after verifying it still contains <oldvalue>.

With `--stdin`, update-ref reads instructions from standard input and performs all modifications together. Specify commands of the form:

```
update SP <ref> SP <newvalue> [SP <oldvalue>] LF
create SP <ref> SP <newvalue> LF
delete SP <ref> [SP <oldvalue>] LF
verify SP <ref> [SP <oldvalue>] LF
option SP <opt> LF
```

With `--create-reflog`, update-ref will create a reflog for each ref even if one would not ordinarily be created.

Quote fields containing whitespace as if they were strings in C source code; i.e., surrounded by double-quotes and with backslash escapes. Use 40 "0" characters or the empty string to specify a zero value. To specify a missing value, omit the value and its preceding SP entirely.

Alternatively, use `-z` to specify in NUL-terminated format, without quoting:

```
update SP <ref> NUL <newvalue> NUL [<oldvalue>] NUL
create SP <ref> NUL <newvalue> NUL
delete SP <ref> NUL [<oldvalue>] NUL
verify SP <ref> NUL [<oldvalue>] NUL
option SP <opt> NUL
```

In this format, use 40 "0" to specify a zero value, and use the empty string to specify a missing value.

In either format, values can be specified in any form that Git recognizes as an object name. Commands in any other format or a repeated <ref> produce an error. Command meanings are:

update

Set <ref> to <newvalue> after verifying <oldvalue>, if given. Specify a zero <newvalue> to ensure the ref does not exist after the update and/or a zero <oldvalue> to make sure the ref does not exist before the update.

create

Create <ref> with <newvalue> after verifying it does not exist. The given <newvalue> may not be zero.

delete

Delete <ref> after verifying it exists with <oldvalue>, if given. If given, <oldvalue> may not be zero.

verify

Verify <ref> against <oldvalue> but do not change it. If <oldvalue> zero or missing, the ref must not exist.

option

Modify behavior of the next command naming a <ref>. The only valid option is `no-deref` to avoid dereferencing a symbolic ref.

If all <ref>s can be locked with matching <oldvalue>s simultaneously, all modifications are performed. Otherwise, no modifications are performed. Note that while each individual <ref> is updated or deleted atomically, a concurrent reader may still see a subset of the modifications.

Logging Updates

If config parameter "core.logAllRefUpdates" is true and the ref is one under "refs/heads/", "refs/remotes/", "refs/notes/", or the symbolic ref HEAD; or the file "$GIT_DIR/logs/<ref>" exists then `git update-ref` will append a line to the log file "$GIT_DIR/logs/<ref>" (dereferencing all symbolic refs before creating the log name) describing the change in ref value. Log lines are formatted as:

1. oldsha1 SP newsha1 SP committer LF

   Where "oldsha1" is the 40 character hexadecimal value previously stored in <ref>, "newsha1" is the 40 character hexadecimal value of <newvalue> and "committer" is the committer’s name, email address and date in the standard Git committer ident format.

Optionally with -m:

1. oldsha1 SP newsha1 SP committer TAB message LF

Where all fields are as described above and "message" is the value supplied to the -m option.

An update will fail (without changing <ref>) if the current user is unable to create a new log file, append to the existing log file or does not have committer information available.

GIT

Part of the git[1] suite

verify-pack

NAME

git-verify-pack - Validate packed Git archive files

SYNOPSIS

```
git verify-pack [-v|--verbose] [-s|--stat-only] [--] <pack>.idx …
```

DESCRIPTION

Reads given idx file for packed Git archive created with the *git pack-objects* command and verifies idx file and the corresponding pack file.

OPTIONS

<pack>.idx …

The idx files to verify.

-v

--verbose

After verifying the pack, show list of objects contained in the pack and a histogram of delta chain length.

-s

--stat-only

Do not verify the pack contents; only show the histogram of delta chain length. With `--verbose`, list of objects is also shown.

--

Do not interpret any more arguments as options.

OUTPUT FORMAT

When specifying the -v option the format used is:

```
SHA-1 type size size-in-packfile offset-in-packfile
```

for objects that are not deltified in the pack, and

```
SHA-1 type size size-in-packfile offset-in-packfile depth base-SHA-1
```

for objects that are deltified.

GIT

Part of the git[1] suite

write-tree

NAME

git-write-tree - Create a tree object from the current index

SYNOPSIS

```
git write-tree [--missing-ok] [--prefix=<prefix>/]
```

DESCRIPTION

Creates a tree object using the current index. The name of the new tree object is printed to standard output.

The index must be in a fully merged state.

Conceptually, *git write-tree* sync()s the current index contents into a set of tree files. In order to have that match what is actually in your directory right now, you need to have done a *git update-index* phase before you did the *git write-tree*.

OPTIONS

--missing-ok

Normally *git write-tree* ensures that the objects referenced by the directory exist in the object database. This option disables this check.

--prefix=<prefix>/

Writes a tree object that represents a subdirectory `<prefix>`. This can be used to write the tree object for a subproject that is in the named subdirectory.

GIT

Part of the git[1] suite